NANTUCKET CLIPPER

5.2

Title: CA-Clipper 5.2e . Guide To CA-Clipper
Credits: The Guide To CA-Clipper, Copyright (C) Computer Associates 1995
Legend: . denotes changed material
. denotes new material
* denotes obsolete items or usage

1. Language

1. Functions
2. Commands
3. Classes
4. Statements
5. Operators
6. Directives
7. Get System
8. Drivers

9. Tables

1. Inkey Codes
2. Colors
3. Picture Codes
4. ASCII Chart
5. Box Characters
6. Special Characters
7. Reserved Words
8. Unsupported dBASE Items

9. Glossary

1. Glossary
2. Categories
3. License Agreement

Abbreviation

(preprocessor) A source token whose leftmost characters exactly match the leftmost characters of a keyword in a translation directive. Abbreviations must be at least four characters in length.

Activation

(procedure and function) The commencement of execution of a procedure. Each call to a procedure is referred to as an activation of that procedure. If the procedure in turn calls another procedure (or calls itself recursively), a new activation is said to have occurred. The earlier activation is then referred to as a pending activation.

Pending activations are often referred to as higher-level activations or higher-level procedures. When one procedure calls another (i.e., creates a new activation), the latter is often referred to as a lower-level activation or a lower-level procedure.

See Also: Activation Record Activation Stack Function Procedure

Activation Record

(procedure and function) An internal data structure that contains information pertaining to an activation.

See Also: Activation Activation Stack

Activation Stack

(procedure and function) An internal data structure that contains an activation record for the current activation and all pending activations.

See Also: Activation Activation Record Stack

Active Window

(debugger) The window to which all keystrokes (except those valid in the Command Window) apply. An active window is indicated by a highlighted border. The Tab and Shift-Tab keys are used to select the next and previous window, respectively.

Algorithm

(algorithm) A set of rules and/or a finite series of steps that will accomplish a particular task.

See Also: Iteration Selection Sequence

Alias

(database) The name of a work area; an alternate name given to a database file. Aliases are often used to give database files descriptive names and are assigned when the database file is opened. If no alias is specified when the database file is USEd, the name of the database file becomes the alias.

An alias can be used to reference both fields and expressions (including user-defined functions). In order to alias an expression, the expression must be enclosed in parentheses.

See Also: Work Area

Animate Mode

(debugger) The mode of execution in which an application runs one line at a time until a Breakpoint or Tracepoint is reached, with the Execution Bar moving to each line as it is executed.

Application

(configuration) A program designed to execute a set of interrelated tasks. Typically referring to a system designed to address a particular business purpose (e.g., Order Entry/Inventory/Invoicing, a document tracking database, or an insurance claims calculator).

Argument

(variable) Generally, a value or variable supplied in a function or procedure call, or an operand supplied to an operator. In function and procedure calls, arguments are often referred to as actual parameters.

See Also: Parameter

Array

(array) A data structure that contains an ordered series of values called elements. The elements of an array are referred to by ordinal number; the first element is number 1, the second is number 2, etc. A numeric expression used to specify an element of an array is referred to as a subscript or index. In CA-Clipper, the elements of an array may be values of any type, including references to other arrays.

See Also: Array Reference Nested Array Subarray Subscript

Array Functions

(array) Those functions that specifically perform their tasks on arrays.

See Also: Array Element Function Subscript

Array Iterator

```
(array) A function that traverses an array, performing an operation on
each element.
```

See Also: Array

Array Reference

(array) A special data value that allows access to an array. In CA-Clipper, program variables and array elements cannot directly contain arrays; they may, however, contain array references. A variable that contains a reference to a particular array is said to refer to that array, and the array's elements may be accessed by applying a subscript to the variable. If the value of a variable containing an array reference is assigned to a second variable, the second variable will contain a copy of the array reference; both variables then refer to the same array, and the array's elements may be accessed by applying a subscript to either variable.

See Also: Array Nested Array Subarray Subscript

ASCII

(general) An acronym for the American Standard Code for Information Interchange, the agreed upon standard for representing characters (alphabetic, symbolic, etc.) in the memory of the computer.

Assignment

(expression) The act of copying a new value into a variable. In CA-Clipper this is done with the simple assignment operators (=) and (:=), or the compound operators (+=, -=, *=, **=).

Attribute

(database) As a formal DBMS term, refers to a column or field in a table or database file.

See Also: Column Field

Background Color

(user interface) The color that appears behind displayed text of another color (the foreground color).

See Also: Foreground Color

Beginning of File

(database) The top of the database file. In CA-Clipper there is no beginning of file area or record. Instead, it is indicated by BOF() returning true (.T.) if an attempt is made to move the record pointer above the first record in the database file or the database file is empty.

Binary

(general) The numerical base upon which computer programs are modeled. Also referred to as base 2, it is a mathematical way of representing the electrical circuitry upon which computers operate. A 0 represents an off state, and a 1 indicates an on state. A combination of eight ons and offs are used to express bytes of program or data in memory.

See Also: Byte

Binary File

(file) A file that contains an unformatted sequence of bytes. Carriage return, linefeed, or end of file characters have no special meaning in a binary file. Binary files include executable files, graphics files, or data files.

See Also: Text File

Binary Operator

(expression) An operator that operates on two operands. For example, the addition operator.

See Also: Operator

Blockify

(preprocessor) To change an expression in the source text into a code block definition. Blockifying is accomplished by surrounding the source text with braces and placing an empty block parameter list (a pair of vertical bars) just inside the braces. When the resulting code block is evaluated at runtime, the original expression will be evaluated.

Blockify Result Marker

(preprocessor) A result marker of the form <{id}>. id must correspond to the name of a match marker. A blockify result marker specifies that the corresponding source text is to be blockified. If the matched source text is a list of expressions, each expression in the list is individually blockified. If no source text matched the corresponding match marker, an empty result is produced.

Branching

(language) Changing the sequence of execution in a program. Execution normally proceeds in sequence from the top of a function or procedure to the bottom. When control is transferred to a statement that is not in sequence, execution is said to have branched.

Breakpoint

(debugger) A point at which an application pauses execution and returns control to the debugger.

Buffer

(general) A temporary data storage location in memory. As an example, a disk input-output buffer is an area of memory that stores data read from the disk to temporary locations while processing it.

Byte

(general) Eight bits of data, the smallest unit of information stored in the computer's memory. As an example, one byte is required to represent one ASCII character.

Calling Program

(procedure and function) The procedure, or user-defined function that transferred control to the currently executing procedure or function. When the current procedure or user-defined function terminates with a RETURN statement, this is where control will return.

Callstack

(debugger) A list containing the names of all pending activations at the current point in an application.

Callstack Window

(debugger) The window in which the Callstack is displayed.

Cell

(database) In a table, a cell is the intersection of a Row and a Column.

See Also: Column Row Table

Character

(general) A letter, digit, punctuation mark, or special symbol stored and processed by the computer.

(data type) A special data type consisting of one or more values in the IBM extended character set. Characters can be grouped together to form strings. The maximum size of a character string in CA-Clipper is 65,534 bytes.

See Also: ASCII String

Character Functions

(expression) Those functions that act upon individual characters or strings of ASCII characters in the performance of their tasks.

See Also: ASCII Function String

Classes

(object-oriented) A class defines the variables contained in an object, and the operations applied when the object receives a message. Every object is an instance of a class and responds to messages of that class. Each object, however, has its own copy of the variables specified in the class definition. New objects are created in CA-Clipper by a calling a special function that begins with the class name followed by the New suffix.

See Also: Instance Variables Messages Methods Objects

Clause

(command) An optional or required section of a CA-Clipper command beginning with a keyword that modifies or enhances the command.

Code Block

(data type) A special data type that refers to a piece of compiled program code. In a program, the source code that specifies the creation of a code block.

Code Window

(debugger) The window in which source code is displayed.

Collision

(network) An attempt by more than one user (typically on a networked, multiuser system) to update a database simultaneously, usually resulting in data corruption. Generally due to poor implementation of the code to support multiuser operations.

See Also: Deadly Embrace LAN Local Area Network

Column

(database) A database term used to describe a field in a table or database file.

(user interface) A numeric value that represents a position on the display screen or on the printed page.

See Also: Field

Command

(command) A statement to be translated by the CA-Clipper preprocessor into source code that will perform a particular operation. All CA-Clipper commands are defined in the standard header file, STD.CH, located in \CLIPPER5\INCLUDE. Also, the preprocessor directives that define a command.

See Also: Header File Statement STD.CH

Command Window

(debugger) The window in which commands are displayed and entered.

Comment

(language) Text in a source program that is ignored by the compiler. Usually used to make descriptive comments about the surrounding source code.

Compiler

(program) A program that translates source code output from the preprocessor into object code. The resulting object file can then be linked to produce an executable program using the linker.

See Also: Linker Object File Program File

Concatenate

(expression) To combine two groups of character data together by placing them in a sequence to form a new string of characters.

See Also: Data Type

Concurrency

(database) The degree to which data can be accessed by more than one user at the same time.

Condition

(command, database, expression) A logical expression that determines whether an operation will take place. With database commands, a logical expression that determines what records are included in an operation. Conditions are specified as arguments of the FOR or WHILE clause.

See Also: Scope

Conditional Compilation

(preprocessor) Selective exclusion by the preprocessor of certain source code. The affected source code is bracketed by the #ifdef and #endif directives. It is excluded if the argument in the #ifdef directive is not a #defined identifier.

Console Input/Output

(user interface) A style of operation of the keyboard and display that emulates a simple typewriter-like interface. Console input echoes each key typed and provides processing for the backspace and return keys. Console output wraps to the next line when the output reaches the right edge of the visible display, and scrolls the display when the output reaches the bottom of the visible display.

See Also: Full-screen Input/Output

Constant

(variable) The representation of an actual value. For example, .T. is a logical constant, string is a character constant, 21 is a numeric constant. There are no date and memo constants.

Constant Array

```
(array)
```

See Also: Literal Array

Control Structure

(language) Any program structure that alters the flow of program control. In CA-Clipper, these include:

- BEGIN SEQUENCE...END
- DO WHILE...ENDDO
- DO CASE...ENDCASE
- FOR...NEXT
- IF...ENDIF

Controlling/Master Index

```
(database) The index currently being used to refer to records by key
value or sequential record movement commands.
```

See Also: Index Natural Order

Conversion Functions

(expression) Generally referring to a category of functions whose purpose is to change one data type to another (e.g., to change a number or a date to a character string).

Cursor

(user interface) An onscreen indicator used to show the current keyboard input focus and is displayed as a block or underline character. The cursor moves in response to characters or control keys typed by the user.

(debugger) The cursor indicates the current line and/or column position in the active window or dialog box. Note that some windows, such as the Monitor Window, do not utilize the cursor. When a window that does not utilize the cursor is active, the cursor appears in the Code Window.

See Also: Highlight Input Focus

Data Independence

(modular programming) The technique of writing a program, function, or procedure in such a way that it will be able to perform its operation on data supplied to it without a built-in description of the data format.

Data Type

(data type) The category of a data value. A data type is distinguished by the set of allowable values for that type, the set of operators that can be applied, and the storage format used to represent these values. In CA-Clipper, the following data types are defined: character, numeric, date, logical, array, object, code block, and NIL. Program variables may contain values of any type. Database field variables are limited to character, numeric, date, logical, and a special type called memo which is treated the same as character.

Database

```
(database) An aggregation of related operational data used by an
application system.  A database can contain one or more data files or
tables.
```

See Also: Field Record Tuple View

Date Functions

(expression) Functions that operate on date values (as opposed to character, numeric or other values).

See Also: Function

Date Type

(data type) A special data type consisting of digits to store year, month, and day values. Operations on date values are based on chronological values.

DBMS

(database) An acronym for the term database management system. A DBMS is a software system that mediates access to a database through a data manipulation language.

Deadly Embrace

(network) In a multiuser database management system, the result of an unforeseen flaw that has allowed a situation wherein two or more simultaneous operations cannot be completed because each has instigated locks on files or records that the other needs to complete its task.

Debugger

(debugger) A tool used to track down errors in a program.

Debugging

(error handling) A phase of software development where errors are identified and fixed.

Declaration

```
(variable) A statement used by the compiler to define a variable,
procedure, or function identifier.  The scope of the declaration is
determined by the position of the declaration statement in the source
file.
```

See Also: Identifier Scope

Decrement

(expression) To decrease a value by a fixed amount, usually one. In CA-Clipper, the decrement operator (--) can be used to decrement a numeric value in a variable.

See Also: Assignment Increment

Define

(preprocessor) To #define an identifier to the preprocessor and optionally specify text to be substituted for occurrences of the identifier.

Delimited File

(file, database) A text file that contains variable-length database records with each record separated by a carriage return/linefeed pair (CHR(13) + CHR(10)) and terminated with an end of file mark (CHR(26)). Each field within a delimited file is variable length, not padded with either leading or trailing spaces, and separated by a comma. Character strings are optionally delimited to allow for embedded commas.

See Also: Database SDF File Text File

Delimiter

(general) A character or other symbol that marks a boundary.

Destination

(expression) The variable or array element to receive data in an assignment.

(general) The work area, file, or device to which data is sent.

See Also: Assignment

Device

(general) Either an actual physical component of the computer system such as printer or a DOS handle that refers to it (e.g., PRN:), or a logical device that behaves and is addressed the same way as a physical device (e.g., a print spooler).

Dialog Box

(debugger) A box displayed from within the debugger whenever further input is required.

Dimension

(array) The maximum number of subscripts required to specify an array element. For example, a two-dimensional array must have two subscripts, a three-dimensional array must have three subscripts and so on.

See Also: Subscript

Directive

(preprocessor) An instruction to the preprocessor. Preprocessor directives must appear on a separate line, and must begin with a hash mark (#). Their scope or effect extends from the point where they are encountered to the end of the source file in which they appear.

BIN2L()
Convert a 32-bit signed integer to a numeric value

Syntax

BIN2L(<cSignedInt>) --> nNumber

Arguments

<cSignedInt> is a character string in the form of a 32-bit signed integer number--least significant byte first. Only the first four characters are used by the function; all others are ignored.

Returns

BIN2L() returns an integer numeric value.

Description

BIN2L() is a low-level file function that is used with FREAD() to convert a four-byte character string formatted as a signed integer to a CA-Clipper numeric data type. This is most useful when you are reading foreign file types and want to read numeric data in its native format.

Examples

. This example opens a database file using low-level file functions and reads the number of records (bytes 4-7). The result is the same as with LASTREC():

```
#include "Fileio.ch"
//
nHandle := FOPEN("Sales.dbf", FO_READ)
// Note:  Sales.dbf contains 84 records
//
// Point to byte 4
FSEEK(nHandle, 4, FS_SET)
//
// Read the number of records
cRecords := SPACE(4)
FREAD(nHandle, @cRecords, 4)
//
? LTRIM(STR(BIN2L(cRecords)))         // Result: 84
FCLOSE(nHandle)
```

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM.

See Also: BIN2I() BIN2W() FREAD() FREADSTR() I2BIN() L2BIN()

Directory

(file) The major operating system facility for cataloging files. A directory contains a list of files and references to child directories (subdirectories), and is identified by name. Directories can be nested forming a hierarchical tree structure. The operating system provides a number of facilities that allow users to create and delete directories.

See Also: Disk File Path Volume

Disk

```
A magnetic storage medium designed for long-term storage.  Disks come in
two varieties: hard disks (fast but fixed) or floppy disks (slow but
removable).  A disk can be partitioned into multiple volumes, each
containing a tree-structured directory system that holds files
accessible by programs.
```

See Also: Directory File Volume

Drive

(file) A disk drive or a letter (normally followed by a colon) that designates a disk drive. On most computers, the letters A and B refer to floppy disk drives; other letters refer to fixed disk drives or logical drives (e.g., fixed disk partitions or network drives).

Dumb Stringify Result Marker

(preprocessor) A result marker of the form #<{id}>. id must correspond to the name of a match marker. A dumb stringify result marker specifies that the corresponding source text is to be enclosed in quotes. If the matched source text constitutes a list of expressions, each expression in the list is individually stringified. If no source text was matched, an empty pair of quotes is produced.

Dynamic

(general) Used generically to refer to data or algorithms that change with time. Often used specifically to describe algorithms that automatically adjust to prevailing conditions.

Dynamic Overlay

(linker) Allows a module's code to be divided into pages to be brought into and out of memory on a least recently used basis.

Dynamic Scoping

(variable) A method of determining an item's existence or visibility based on the state of a program during execution. Example: A CA-Clipper public variable may or may not be visible within a particular function, depending on whether the variable has been created and whether a previously called function has obscured it by creating a private variable with the same name.

See Also: Lexical Scoping Scope

Element

(array) A component unit of an array, usually referred to by a numeric subscript or index.

See Also: Array Subscript

Empty Result

(preprocessor) An absence of result text; the effect of certain result markers when the corresponding match marker did not match any source text (but when the translation directive as a whole was matched). An empty result simply implies that no result text is written to output.

Encapsulation

(modular programming) Generically, the design of a function or program that obeys the principle of information hiding. A function or program is said to be encapsulated if other programs or functions have no knowledge of its inner workings. A data structure is said to be encapsulated if knowledge of its internal organization is limited to a single function or module.

See Also: Information Hiding Lexical Scoping Modularity Side Effect

End of File

(database) The bottom of a database file. In CA-Clipper, this is LASTREC() + 1 and is indicated by EOF() returning true (.T.).

Enhanced Color

(user interface) The color used to display GETs or PROMPTs (if INTENSITY is ON).

See Also: Standard Color

Environment Variables

(configuration) Operating system variables that can be used to communicate configuration information to executable programs. Environment variables are manipulated using the DOS SET command. The CA-Clipper compiler and linker respond to certain environment variables. CA-Clipper programs can inspect the settings of environment variables using the GETENV() function.

Error

(error handling) The presence of some element of an operation that does not satisfy the requirements of the operation. An error when encountered causes failure, which in turn raises an exception.

See Also: Exception Failure Runtime Error

Error Handling

(error handling) The concept of including code in a program so that exceptions to normal operational states that occur during the program execution can be anticipated and dispatched with the least possible detrimental consequences to the use of the program and the data being worked on.

See Also: Exception Runtime Error

Evaluate

(expression) To execute part of a program in order to produce a value. For an expression, to execute the program code associated with the expression and return the resulting value. For the macro operator, to compile the macro string, execute the resulting program code, and return the resulting value.

See Also: Expression

Exception

(error handling) An occurrence of an abnormal condition during the execution of an operation. An exception is said to be raised when an operation fails.

See Also: Error Failure Runtime Error

Exclusive

(network) In a network, to assure that no other user will write data to a file, it may be opened in an Exclusive mode. Only the user opening the file exclusively may then access it until exclusive use of the file (by closing it, or opening it SHARED) is relinquished.

Executable File

(configuration) A file output from the linker directly executable from the operating system command line. Executable files have an .EXE extension.

See Also: Linker

Execution Bar

(debugger) The highlight bar which is positioned on the line of code to be executed next.

Exported Instance Variables

```
(object-oriented)
```

See Also: Instance Variables

Expression

(expression) A combination of constants, identifiers, operators, and functions that yield a single value when evaluated.

Extension

(file) A filename extension normally used for identifying the type or originating program of a file.

See Also: Drive Filename Path

Failure

```
(error handling) The inability of an operation to satisfy its purpose.
When a failure occurs an exception is raised.  Failures are due in large
part to errors.
```

See Also: Error Exception Retry Runtime Error

Field

(database) The basic column unit of a database file. A field has four attributes: name, type, length, and decimals if the type is numeric.

See Also: Database Record Tuple Vector View

Field Variable

```
(variable, database) A variable that refers to data in a database field,
as opposed to data in memory.
```

See Also: Local Variable Memory Variable Variable

File

```
(file) A file is an organized collection of bytes stored on disk,
maintained by the operating system, and referenced by name.  Its
internal structure is solely determined by its creator.
```

See Also: Binary File Database Text File

File Handle

(file) An integer numeric value returned from FOPEN() or FCREATE() when a file is opened or created. This value is used to identify the file for other operations until it is closed.

File Locking

```
(network) The process by which a user is guaranteed exclusive access to
a database file.  The file is only available to the user that applied
the lock.
```

See Also: Record Locking

File Server

(network) A computer on a network dedicated to providing data storage to other computers (i.e., workstations) for the purpose of sharing information among multiple users. File servers tend to provide other services such as E-Mail service and shared printer support as well.

Filewide Declaration

(variable) A variable declaration statement that has the scope of the entire source file. Filewide declarations are specified before the first procedure or function declaration in a program file and the program file must be compiled with the /N option.

See Also: Scope Storage Class

Filename

(file) The name of a disk file that may optionally include a drive designator, path, and extension.

See Also: Drive Extension Path

Foreground Color

(user interface) The color of text appearing on the screen, usually on a different colored background.

See Also: Background Color

Formfeed

```
(general) A special character (CHR(12)) that by convention causes most
printers to move the printhead to the top of the next page.
```

See Also: Hard Carriage Return Linefeed

Freeformat

(linker) The suggested command interface for .RTLink, allowing you to specify linker commands in any order on the command line. It is easier to create, examine and change FREEFORMAT command lines. It is compatible with the Plink86-Plus syntax.

Full-screen Input/Output

(user interface) A style of operation of the keyboard and display used for complex data entry and display tasks. Full-screen input and output are generally performed using the @..SAY, @..GET and READ commands. Full-screen output is distinguished from console-style output by the fact that control characters (e.g., backspace, carriage return) are not processed, and wrapping and scrolling do not occur at the boundaries of the visible display area.

See Also: Console Input/Output

Function

(procedure and function) An executable block of code with an assigned name. Alternately, the collection of source code statements that define a function. Certain functions are supplied as part of CA-Clipper; others are defined by the programmer using the FUNCTION or PROCEDURE declaration statements. The latter are referred to as user-defined functions.

The terms procedure and function are generally interchangeable. By convention, a function returns a value, while a procedure does not.

See Also: Activation Parameter Procedure

Group

(linker) An Intel 8086 addressing classification defining a collection of segments to be addressed using the same segment register. Note that a group is not a section but rather a logical concept used only for addressing.

Hard Carriage Return

(general) An explicit carriage return character at the end of a line in a text file, as opposed to a soft carriage return that might be inserted into text by a program designed to handle word wrapping. A hard carriage return character is generated by the expression (CHR(13)) where a soft carriage return character is generated with the expression (CHR(141)).

Header File

(configuration, preprocessor) A source file containing manifest constant definitions; command or pseudofunctions; and/or program statements merged into another source file using the #include preprocessor directive.

See Also: Program File Source Code STD.CH

Help Window

(debugger) The window in which online help is displayed.

Hexadecimal

(general) A representation of a value in base 16 rather than decimal which is base 10. Hexadecimal values are easily converted to and from binary (base 2) which is the form of data the computer actually uses. Hexadecimal values are represented by digits zero through nine and A through F for values between 10 and 15.

Hidden

```
(modular programming) The resulting state of a module of a program
written to conform to the principals of information hiding.
```

See Also: Information Hiding

Highlight

(user interface) Indicates input focus for menus, browsers, or GETs. With menus and browsers, the currently selected item or cell has input focus and is displayed in the current enhanced color or inverse video. With GETs, the current GET is highlighted in the current enhanced color or inverse video while the other GETs are displayed in the current standard color if an unselected color setting is active.

See Also: Cell Enhanced Color Input Focus Standard Color

IBM Extended Character Set

(general) The character set built into the ROM of the IBM PC. This character set is a superset of ASCII, containing additional special characters (such as a line drawing character set) that may be used to enhance your program screens.

Identifier

(preprocessor, procedure and function, variable) A name that identifies a function, procedure, variable, constant or other named entity in a source program. In CA-Clipper, identifiers must begin with an alphabetic character and may contain alphabetic characters, numeric characters, and the underscore character.

Include File

(preprocessor)

See Also: Header File

Increment

(expression) To increase a value by a fixed amount, usually one. In CA-Clipper the increment operator (++) can be used to increment a numeric value in a variable.

Incremental Linking

(linker) The ability to link only the modules of an application that have been changed, greatly increasing the speed in which the link occurs.

See Also: Linking Module

Index

(database) An ordered set of key values that provides a logical ordering of the records in an associated database file. Each key in an index is associated with a particular record in the database file. The records can be processed sequentially in key order, and any record can be located by performing a SEEK operation with the associated key value.

See Also: Controlling/Master Index Key Value Natural Order

Information Hiding

(procedure and function) A fundamental programming principle that states that functions and programs should conceal their inner workings from other functions and programs. Stated simply: a function should possess only the knowledge necessary for it to accomplish its task. When one function calls another, the calling function should possess only the knowledge explicitly required to call the other function.

See Also: Encapsulation Lexical Scoping Modularity Side Effect

Initialize

(variable) To assign a starting value to a variable. If initialization is specified as part of a declaration or variable creation statement, the value to be assigned is called an initializer.

See Also: Assignment

Input Focus

(user interface) The GET, browse cell, or menu item where user interaction can take place is said to have input focus. The item with input focus usually is displayed in enhanced color or inverse video.

Insert Mode

(user interface) A data entry mode entered when the user presses the insert key. When this mode is active, characters are inserted at the cursor position. Text to the right of the cursor is shifted right.

See Also: Overstrike Mode

Inspecting

(debugger) The process of examining work areas, variables, expressions and activations inside the debugger.

Instance Variables

(object-oriented) Instance variables are the attributes or the data portion of an object as defined by the object's class. Each object, when created, is given its own unique set of instance variables initialized to their default values. Instance variables that are accessible are called exported instance variables. Exported instance variables can be inspected and--in some cases--assigned using the send operator. The instance variables of an object persist as long as the object it belongs to.

See Also: Classes Objects

Integer

(data type) A number with no decimal digits. Note that CA-Clipper does not provide a separate data type for integer values.

Iteration

(algorithm) One of the three basic building blocks of algorithm development (the others are sequence and selection). Iteration refers to operations that are performed repeatedly, usually until some condition is satisfied.

See Also: Selection Sequence

Join

(database) An operation that takes two tables as operands and produces one table as a result. It is, in fact, a combination of other operations including selection and projection.

See Also: Projection Selection

Key Expression

(database) An expression, typically based on one or more database fields, that when evaluated, yields a key value for a database record. Key expressions are most often used to create indexes or for summarization operations.

See Also: Index Key Value

Key Value

(database) The value produced by evaluating a key expression. When placed in an index, a key value identifies the logical position of the associated record in its database file.

See Also: Index Key Expression

Keyboard Buffer

(user interface) An area of memory dedicated to storing input from the keyboard while a program is unable to process the input. When the program is able to accept the input, the keyboard buffer is emptied.

Keyword

(command, language) A word that has a special meaning to a compiler or other utility program. Commands, directives, or options are often recognized by examining supplied text to see if it contains keywords.

LAN

(network) An acronym for Local Area Network. Generally used to describe a system by which microcomputers are connected together to perform such functions as file and peripheral sharing, electronic mail, and centralized backup of data.

Lexical Scoping

```
(variable) A method of determining an item's existence, visibility, or
applicability (i.e., the item's scope) by it's position within the text
of a program.
```

See Also: Local Variable Scope Static Variable

Lexically Scoped Variable

(variable) A variable that is only accessible in a particular section of a program, where that section is defined using simple textual rules. For example, a local variable is only accessible within the procedure that declares it.

See Also: Dynamic Scoping Local Variable Static Variable

Library

(linker) A file containing one or more object modules. Modules are extracted by linker and combined with object files to form an executable (.EXE) file or a prelink library (.PLL) file.

Library File

(configuration) A file containing one or more object modules. The linker searches specified libraries to resolve references to functions or procedures that were not defined in the object files being linked.

See Also: Linker Module Object File

Lifetime of a Variable

```
(variable) The period of time during which a variable retains its
assigned value.  The lifetime of a variable depends on its storage
class.
```

See Also: Scope Visibility

Linefeed

(general) A special character (CHR(10)) that by convention causes the cursor or printhead to move to the next line or to terminate a line in a text file. It is usually used in combination with a hard carriage return.

See Also: Formfeed Hard Carriage Return

Linker

(program) A program that combines object files created by a compiler to produce an executable program. The linker examines the supplied object files to resolve symbol references between modules. If a module refers to a symbol that is not defined by any of the modules, the linker searches one or more libraries to resolve the reference.

See Also: Library File Object File

Linking

(linker) The process in which object files and libraries are combined and references are resolved to produce a relocatable memory image (generally, an executable).

List

(expression, command) A list of expressions, field names, or filenames, separated by commas specified generally as command, procedure, or function arguments. Code blocks can also execute a list of expressions.

List Match Marker

(preprocessor) A match marker indicating a position that will successfully match a list of one or more arbitrarily complex expressions. A list match marker marks a part of a command that is expected to consist of a list of programmer-supplied expressions. A list match marker has the form <id,...>. id associates a name with the match marker. The name can be used in a result marker to specify how the matching source text is to be handled.

Literal

```
(data type) A source code element interpreted literally (as
encountered), and assumed to have no abstract meaning.  Generally a
constant.
```

See Also: Constant

Literal Array

(array) In CA-Clipper, an array specified by enclosing a series of expressions in curly ({}) braces. A literal array is an expression that evaluates to an array reference.

See Also: Array Array Reference

Local Area Network

(network) A system by which microcomputers are connected (via coaxial cable, optical fiber, twisted pair phone wire, or other media), relying on sophisticated operating software to perform such functions as file and peripheral sharing, electronic mail, and centralized backup of data.

See Also: LAN

Local Variable

(variable) A variable that exists and retains its value only as long as the procedure in which it is declared is active (i.e., until the procedure returns control to a higher-level procedure). Local variables are lexically scoped; they are accessible by name only within the procedure where they are declared.

See Also: Dynamic Scoping Lexical Scoping Static Variable

Logical Type

```
(data type) A special data type consisting of true (.T.) or false (.F.)
values.
```

See Also: Condition

Logify

(preprocessor) To change an expression in the source text into a logical value. Logifying is accomplished by surrounding the expression with periods.

Logify Result Marker

(preprocessor) A result marker of the form #<.id.>. id must correspond to the name of a match marker. This result marker writes true (.T.) to the result text if any input text is matched; otherwise, it writes false (.F.) to the result text. The input text itself is not written to the result text.

BIN2W()
Convert a 16-bit unsigned integer to a numeric value

Syntax

BIN2W(<cUnsignedInt>) --> nNumber

Arguments

<cUnsignedInt> is a character string in the form of a 16-bit unsigned integer number--least significant byte first. Only the first two characters are used by the function; all others are ignored.

Returns

BIN2W() returns an integer numeric value.

Description

BIN2W() is a low-level file function that is used with FREAD() to convert a two-byte character string formatted as an unsigned integer to a CA-Clipper numeric data type. This is most useful when you are reading from a binary file and want to read data in its native format.

Examples

. This example opens a database file using low-level file functions and reads the number of bytes per record (bytes 10-11). The result is the same as with RECSIZE():

```
#include "Fileio.ch"
//
nHandle := FOPEN("Sales.dbf", FO_READ)
// Note:  The length of a record in Sales.dbf is 124
//
// Point to byte 10, the first record size byte
FSEEK(nHandle, 10, FS_SET)
//
// Read record size
cRecSize := SPACE(2)
FREAD(nHandle, @cRecSize, 2)
//
? LTRIM(STR(BIN2W(cRecSize)))            // Result: 124
FCLOSE(nHandle)
```

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM.

See Also: BIN2I() BIN2L() FREAD() FREADSTR() I2BIN() L2BIN()

Macro

(expression) In CA-Clipper, an operation that allows source code to be compiled and executed at runtime. In CA-Clipper, the macro symbol (&) does not perform text substitution unless embedded within a character string. Instead, it is generally treated as a unary operator that operates on a character string. The text in the character string is compiled on the fly using a special runtime compiler. The resulting code is then executed, and the value obtained is returned as the result of the macro operation.

See Also: Code Block Unary Operator

Make

(program) A program used to maintain multifile program systems. A make program takes as its input a file (make file) specifying the relationships between files. When executed the make program compares the date and time stamps of specified target files to the specified dependent files. If any of the dependent files have a more recent date and time stamp than the associated target files, a series of actions are performed.

See Also: Make File

Make File

(configuration) A text file used as input to a make utility containing the specifications and actions required to build a program or a system of programs. This file is often referred to as a description file.

See Also: Make

Manifest Constant

(preprocessor) An identifier specified in a #define directive. The preprocessor substitutes the specified result text whenever it encounters the identifier in the source text.

Map File (.MAP)

(linker) The map file (.MAP) contains information about symbol and segment addresses within the memory image created by .RTLink. It is generated when requested through the use of the appropriate command line switch. A map file generated during a link will have much more information than one produced during the creation of a prelinked library. During the .PLL creation, only symbols, names, and some relative addresses are known. During a link, the final memory layout is known, and a more detailed map can be created.

Master Index

(database)

See Also: Controlling/Master Index

Match

(preprocessor) A successful comparison of source text with a match pattern (or part of a match pattern).

Match Marker

(preprocessor) A construct used in a match pattern to indicate a position that will successfully match a particular type of source text. There are several types of match markers, each of which will successfully match a particular type of source text.

Match Pattern

(preprocessor) The part of a translation directive that specifies the format of source text to be affected by the directive. A match pattern generally consists of words and match markers.

Memo Type

(data type, database) A special database field type consisting of one or more characters in the IBM extended character set. The maximum size of a memo field in CA-Clipper is 65,534 bytes. A memo field differs only from a character string by the fact it is stored in a separate memo (.DBT file) and the field length is variable-length

See Also: Character String

Memory Variable

```
(variable) In general, a variable that resides in memory, as opposed to
a database field variable.  Sometimes used specifically to refer to
variables of the MEMVAR storage class (private and public variables), as
opposed to static or local variables.
```

See Also: Field Variable Private Variable Public Variable Variable

Menu

(user interface) An onscreen list of choices from which the user selects. Menus range from simple to elaborate forms. Two examples are menus that pull down from the top of the screen (an elaborate type requiring more programming), or a simple list of numbered items from which the user selects by entering the appropriate number.

Menu Bar

(debugger) The bar at the top of the debugger screen, on which the available menu choices are displayed.

Messages

(object-oriented) A message is the way an object is requested to perform some action. Messages are sent to an object and composed of the object name, the send operator, and the selector name followed by arguments enclosed in parentheses. The selector has the same name as the method it is calling. Sending a message produces a return value, much like a function call with the return value varying depending on the operation performed.

See Also: Instance Variables Methods Objects

Metasymbol

(language) Descriptive symbols used in syntax to represent information that must be supplied as part of a source code statement. A metasymbol is constructed using two information components: a data type prefix and a logical descriptor.

Methods

```
(object-oriented) A method is the operation performed in response to a
message sent to an object.
```

See Also: Classes Messages Objects

Modularity

(modular programming) Roughly, a measure of a system's adherence to the principles of modular programming. The principles of modular programming are not precisely defined, but may be said to comprise these basic ideas: programs should be organized as well-defined modules; modules should correspond with syntactic units of the programming language (such as functions or source files); a module should accomplish a well-defined task; a module should interact with as few other modules as possible; interactions between modules should be explicitly specified in the source code for the modules; modules should obey the principle of information hiding.

See Also: Encapsulation Information Hiding Lexical Scoping Side Effect

Module

(modular programming) Generically, a procedure or function (or a set of related procedures and functions) that can be treated as a unit. Sometimes used to refer specifically to the code in a single object file, normally the result of compiling a single source file.

(linker) A portion of the object code that is a discrete unit. If any part of a module is linked, the entire module must be linked.

See Also: Object File Program File

Monitor Window

(debugger) The window in which monitored variables are displayed.

Monitored Variable

(debugger) A variable which is selected by the options on the Monitor Menu and displayed in the Monitor Window.

Multidimensional Array

(array) In CA-Clipper, an array whose elements consist entirely of references to other arrays (called subarrays). The elements of the subarrays may, in turn, contain references to other arrays. Arrays organized in this fashion are said to be nested. Each level of nesting may be viewed as a dimension of the main array, and the elements of the subarrays may be accessed by applying multiple subscripts to the main array.

See Also: Array Array Reference Nested Array Subscript

Natural Order

(database) For a database file, the order determined by the sequence in which records were originally entered into the file. Also called unindexed order.

See Also: Index

Nested Array

(array) In CA-Clipper, two arrays are said to be nested if one of them contains a reference to the other. When an array contains a reference to a second array, the second array is sometimes called a subarray of the first array.

See Also: Array Array Reference Multidimensional Array

NIL

(data type) A special data type that has only one allowable value. The special value (NIL) is automatically assigned to all uninitialized variables except publics, and is also passed as a substitute when arguments are omitted in a procedure or function call.

Normal Stringify Result Marker

(preprocessor) A result marker of the form <"id">. id must correspond to the name of a match marker. A normal stringify result marker specifies that the corresponding source text is to be enclosed in quotes. If the matched source text is a list, each element of the list is individually stringified. If no source text was matched, an empty result is produced.

Normalization

(database) The process of elimination and consolidation of redundant data elements in a database system.

Numeric Type

(data type) A special data type consisting of values that indicate magnitude. Numeric values consist of digits between zero and nine, a sign, and a decimal point.

Object File

(configuration) A file that contains the output of a compiler or other language translator, generally the result of compiling a single source file. Object files are linked to create an executable program.

See Also: Linking Program File

Objects

(object-oriented, data type) An object is an instance of a class. Each object has one or more attributes (called instance variables) and a series of operations (methods) that execute when a message is sent to the object. The object's instance variables can only be accessed or assigned by sending messages to the object. Objects are created by calling a special function associated with a class.

See Also: Classes Instance Variables Messages Methods

Operand

(expression) A value that is operated on by an operator, or the term in an expression that specifies such a value. For example, in the expression x + 5, x and 5 are operands.

See Also: Operator

Operating System

The basic software program that organizes and services the computer and its peripheral devices. The operating system supported by CA-Clipper, MS/PC-DOS is organized into several layers as follows:

- Loader is the layer which brings the operating system software into memory.

- BIOS is the basic hardware interface layer that provides services to the kernel and consists of initialization code and device drivers.

- Kernel is the application interface layer and provides services for process control, memory management, peripheral support, and a file system.

- User interface shell (COMMAND.COM) provides basic services to the user including an interactive mode, directory management, and a service for loading and executing application programs.

- Support programs provide extended operating services not resident in the user interface shell.

Operator

(expression) A symbol that identifies a basic operation. For example, the multiplication operator (*) denotes that two values are to be multiplied. Operators are categorized as either unary or binary, depending on whether they require one or two operands, respectively.

See Also: Binary Operator Operand Unary Operator

Optional Clause

(command, preprocessor) A portion of a match pattern that is enclosed in square ([ ]) brackets. An optional clause specifies part of a match pattern that need not be present for source text to match the pattern. An optional clause may contain any of the components legal within a match pattern, including other optional clauses. When a match pattern contains a series of optional clauses that are immediately adjacent to each other, the matching portions of the source text are not required to appear in the same order as the clauses in the match pattern. If an optional clause is matched by more than one part of the source text, the multiple matches may be handled using a repeating clause in the result pattern.

Overlay

(linker) A section of an executable program that shares memory with other sections of the same program. An overlay is read into memory when the code residing in it is requested by the root (nonoverlayed) section or another overlay.

See Also: Dynamic Overlay

Overstrike Mode

(user interface) A data entry mode entered when the user presses the insert key. When this mode is active, characters are entered at the cursor position and text to the right of the cursor remains stationary.

See Also: Insert Mode

Parameter

(variable) A identifier that receives a value or reference passed to a procedure or user-defined function. A parameter is sometimes referred to as a formal parameter.

See Also: Activation Argument Function Procedure Reference

Path

(file) A literal string that specifies the location of a disk directory in the tree structured directory system. A path specification consists of the following elements: an optional disk drive letter followed by a colon, an optional backslash indicating that the path starts at the root directory of the specified drive, the names of all the directories from the root directory to the target directory, separated by backslash (\) characters. Example: C:\CLIPPER5\INCLUDE. A path list is a series of path specifications separated by semicolons.

Picture

(user interface) A string that defines the format for data entry or display in a GET, SAY, or the return value of TRANSFORM(). Picture strings are comprised of functions which affect the formatting as a whole and a series of template characters that affect formatting on a character by character basis.

See Also: Template

Port

(general) A designation for the hardware that allows the processor to communicate with peripheral devices.

Positional

(linker) The POSITIONAL command interface requires that certain items appear on the input line in a specific order. This syntax is similar to Microsoft LINK interface. Because this syntax limits the use of .RTLink overlays to one overlay area, it is recommended that the FREEFORMAT syntax be used.

Prelinked Library

(linker) Part of the executable program that is stored external to the .EXE file. Prelinked libraries are created before producing an executable in a multistep link. This allows you to create a runtime library with code that you access from different programs, considerably speeding up the linking process.

Preprocessor

(program, preprocessor) A translation program that prepares source code for compilation by applying selective text replacements. The replacements to be made are specified by directives in the source file. In CA-Clipper, the preprocessor operates transparently as a part of the compiler program (CLIPPER.EXE).

See Also: Compiler

Precedence

(expression) The stature of an operator in the hierarchy that determines the order in which expressions are evaluated. For example, the expression 5 + 2 * 3 is interpreted as 5 + (2 * 3) because the multiply operator (*) has a higher precedence than the addition operator (+).

See Also: Expression

Primitive

(procedure and function) A simple, low-level function used by other high-level functions or programs to perform a more complex task.

Print Spooler

A program running either on a local workstation or on the file server that captures print jobs to a file and then queues them for printing later. Print spoolers generally operate as background tasks in order to facilitate printing while other tasks are operating in the foreground.

Private Variable

```
(variable) A variable of the MEMVAR storage class.  Private variables
are created dynamically at runtime using the PRIVATE statement, and
accessible within the creating procedure and any lower-level procedures
unless obscured by another private variable with the same name.
```

See Also: Activation Dynamic Scoping Function Public Variable

Procedure

(procedure and function) An executable block of code with an assigned name. Alternately, the collection of source code statements that define a procedure. In CA-Clipper, this can be a source file (.prg), a format file (.fmt), an explicitly declared procedure (PROCEDURE), or an explicitly declared function (FUNCTION).

The terms procedure and function are generally interchangeable. By convention, a function returns a value, while a procedure does not.

See Also: Activation Function Parameter

Procedure File

(configuration) An ASCII text file containing CA-Clipper procedure and function definitions usually ending with a (.prg) extension; a program file.

See Also: Program File

Program Editor

(program) A program that creates and edits text files or programs.

Program File

(configuration) An ASCII text file containing CA-Clipper source code. Program files usually end with a (.prg) extension. The compiler reads the program file, translates the source code, and produces an object file, that is then linked to produce an executable program.

See Also: Linking Object File Source Code

Projection

(database) A DBMS term specifying a subset of fields. In CA-Clipper, the analogy is the FIELDS clause.

See Also: Join Selection

Prompt

(user interface) A series of characters displayed on the screen indicating that input from the keyboard is expected.

Pseudofunction

(preprocessor) A function-like construct that is replaced with another expression via the #define directive, rather than compiled into a conventional function call. Pseudofunctions may contain parenthesized arguments that may be included in the substituted text.

Public Variable

(variable) A variable of the MEMVAR storage class. Public variables are created dynamically at runtime using the PUBLIC statement, and are accessible from any procedure at any level unless obscured by a private variable with the same name.

See Also: Activation Dynamic Scoping Function Private Variable

Query

(database) A request for information to be retrieved from a database. Alternately, a data structure in which such a request is encoded.

(general) A query is also a general term used when you want to interrogate a setting or an exported instance variable for its current value.

Queue

```
(general) A data structure of variable length where elements are added
to one end and retrieved from the other.  A queue is often described as
first in, first out.
```

See Also: Print Spooler Stack

Record

(database) The basic row unit of a database file consisting of one or more field elements.

See Also: Database Field Table Tuple

Record Locking

(network) The process by which one user obtains exclusive access to a record in a database to prevent another user from attempting to write data to it concurrently. A record lock must be applied prior to writing to a database in use by more than one user.

Recovery

(error handling) The process of attempting to handle an exception or runtime error. Generally, recovery consists of three possible actions: terminate processing, retry the failed operation, or resume processing with the next operation. In all cases, the environment of the program must be restored to a stable state.

See Also: Error Exception Failure Retry Runtime Error

Recursion

(expression) The calling of a procedure by a statement in that same procedure. When a procedure calls itself it is said to recurse. A recursive call causes a new activation of the procedure. If the source code for the procedure includes a declaration of local variables, a new set of local variables is created for each activation. A private variable created by the procedure is associated with the activation in which it is created, and is visible in that activation and any lower-level activations, unless obscured by a private variable created in a lower-level activation.

See Also: Activation Function Private Variable Procedure

Reference

(array, variable) A special value that refers indirectly to a variable or array. If one variable contains a reference to a second variable (achieved by passing the second variable by reference in a function or procedure call), operations on the first variable (including assignment) are passed through to the second variable. If a variable contains a reference to an array, the elements of the array can be accessed by applying a subscript to the variable.

See Also: Array Reference Parameter

Regular Match Marker

(preprocessor) A match marker indicating a position that will successfully match an arbitrarily complex expression in the source text. A regular match marker generally marks a part of a command that is expected to consist of arbitrary programmer-supplied text, as opposed to a keyword or other restrictive component. In order for the source text to match, it must constitute a properly formed expression. A regular match marker has the form <id>. id associates a name with the match marker. The name can be used in a result marker to specify how the matching source text is to be handled.

Regular Result Marker

(preprocessor) A result marker of the form <id>. id must correspond to the name of a match marker. This result marker writes the matched input text to the result text, or nothing if no input text is matched.

Relation

(database) A link between database files that allows the record pointer to move in more than one database file based on the value of a common field or expression. This allows information to be accessed from more than one database file at a time.

Relational Database System

(database) A system that stores data in rows and columns, without system dependencies within the data. In other words, relationships between different databases are not stored in the actual database itself, as is the case in a system that uses record pointers.

Relative Addressing

(user interface) To refer to a memory address, array element, screen location, or printer location with respect to another value, rather than referring to a specific address or element.

Repeating Clause

(preprocessor) A portion of a result pattern surrounded by square ([ ]) brackets. The text specified by the repeating clause is written to output once for each successfully matched match marker in the corresponding match pattern.

Restricted Match Marker

(preprocessor) A match marker indicating a position that will successfully match one or more specified keywords. A restricted match marker marks a part of a command that is expected to be a keyword. A restricted match marker has the form <id: wordList> where wordList is a list of one or more keywords. Source text is successfully matched only if it matches one of the keywords (or is an acceptable abbreviation). id associates a name with the match marker. The name can be used in a result marker to specify how the matching source text is to be handled.

BOF()
Determine when beginning of file is encountered

Syntax

BOF() --> lBoundary

Returns

BOF() returns true (.T.) after an attempt to SKIP backward beyond the first logical record in a database file; otherwise, it returns false (.F.). If there is no database file open in the current work area, BOF() returns false (.F.). If the current database file contains no records, BOF() returns true (.T.).

Description

BOF() is a database function used to test for a boundary condition when you are moving the record pointer backward through a database file using the SKIP command. A simple usage example is a descending order record list with an ascending order index file. A more sophisticated example is a screen paging routine that pages forward or backward through the current database file based on the key the user presses. When the user attempts to page backward, you would use BOF() to test for a beginning of file condition before using the SKIP command to move the record pointer and repaint the screen.

Once BOF() is set to true (.T.), it retains its value until there is another attempt to move the record pointer.

By default, BOF() operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression (see example below).

The SKIP command is the only record movement command that can set BOF() to true (.T.).

Examples

. This example demonstrates BOF() by attempting to move the record pointer before the first record:

```
USE Sales NEW
? RECNO(), BOF()              // Result: 1 .F.
SKIP -1
? RECNO(), BOF()              // Result: 1 .T.
```

. This example uses aliased expressions to query the value of BOF() in unselected work areas:

```
USE Sales NEW
USE Customer NEW
USE Invoices NEW
? Sales->(BOF()), Customer->(BOF())
```

Files: Library is CLIPPER.LIB.

See Also: EOF() SKIP

Result Text

(preprocessor) The text that results from formatting matched source text using a result pattern. If the result text matches a match pattern in another preprocessor directive, then it becomes the source text for that directive. Otherwise, the result text is passed as input to the compiler.

Result Pattern

(preprocessor) The part of a translation directive that specifies the text to be substituted for source text that matches the match pattern. A result pattern generally consists of operators and result markers.

Retry

(network) Upon failing to lock a record in a database that is opened in shared mode, retry refers to attempting the write again based upon certain programmatic parameters.

(error handling) After an exception has been raised and the conditions of a failure corrected, an attempt is made to reexecute the failed operation.

See Also: Error Exception Exclusive Failure Runtime Error

Return Value

(procedure and function) The value or reference returned by a function or method from a function call or message send.

Root

(linker) A special section of the program that has the lowest address of all sections. This is the first section of the program loaded into memory by DOS or the RTLINKST.COM startup code. Other sections are loaded by the overlay manager.

Row

(database) A group of related column or field values that are treated as a single entity. It is the same as a CA-Clipper record.

(user interface) A numeric expression that evaluates to an integer identifying a screen or printer row position.

See Also: Column Field Record

Run Mode

(debugger) The mode of execution in which an application executes without pausing, until a Breakpoint or Tracepoint is reached.

Runtime Error

(error handling) An error that halts a program while it is executing.

Scope

(command, database) In a database command, a clause that specifies a range of database records to be addressed by the command. The scope clause uses the qualifiers ALL, NEXT, RECORD, and REST to define the record scope.

See Also: Condition

Scoreboard

(user interface) An area of the display on line 0 beginning at column 60 that displays status information during certain data entry operations.

Script File

(configuration) A text file that contains command input to a compiler, linker, or other utility program. A script file is often used in lieu of equivalent keyboard input. For the CA-Clipper compiler, script files contain a list of source files to be compiled into a single object file.

(debugger) A file in which frequently used debugger commands are stored and from which those commands can be executed.

Scrolling

(user interface) The action that takes place when the user attempts to move the cursor or highlight beyond the window boundary to access information not currently displayed.

See Also: Window

SDF File

(file, database) A text file that contains fixed-length database records with each record separated by a carriage return/linefeed pair (CHR(13) + CHR(10)) and terminated with an end of file mark (CHR(26)). Each field within an SDF file is fixed-length with character strings padded with trailing spaces and numeric values padded with leading spaces. There are no field separators.

See Also: Database Delimited File Text File

Search Condition

(database)

See Also: Condition Scope

Section

(linker) Load module portion of an .EXE or .OVL file loaded into memory as a single unit. In a program with overlays, the root section containing the main program module loads when the program is executed. Other sections are loaded as overlays when modules within them are invoked.

See Also: Dynamic Overlay Linking

Segment

(linker) Code or data handled by the linker as a indivisible unit.

Selection

(algorithm) One of the three basic building blocks of algorithm development (the others are sequence and iteration). Selection allows control to flow along a number of possible paths, depending on the circumstance encountered.

(database) A DBMS term that specifies a subset of records meeting a condition. The selection itself is obtained with a selection operator. In CA-Clipper, the analogy is the FOR clause.

See Also: Iteration Join Projection Sequence

Self

(object-oriented) An object-oriented term describing a reference to the object that received the current message. In CA-Clipper, this reference is often the return value of a message send.

Send Operator

(object-oriented) A new operator (:) in CA-Clipper used to send messages to user interface objects.

Separator

(file, database) The character or set of characters that differentiate fields or records from one another. In CA-Clipper, the DELIMITED and SDF file types have separators. The DELIMITED file uses a comma as the field separator and a carriage return/linefeed pair as the record separator. The SDF file type has no field separator, but also uses a carriage return/linefeed pair as the record separator.

See Also: Delimiter

Sequence

(algorithm) One of the three basic building blocks of algorithm development (the others are selection and iteration). A sequence is a series of discrete steps that must be performed in a particular order.

(language) In CA-Clipper, a series of statements enclosed in a BEGIN SEQUENCE control structure.

See Also: Algorithm Iteration Selection

Set Colors Window

(debugger) The window in which the Debugger color settings can be inspected.

Shared

(network) A mode in which a file is opened that allows it to be accessed by more than one user at the same time. The inverse of Exclusive.

Shortcutting

(expression) A compiler optimization that causes expressions to be evaluated only to the extent required to determine their outcome. For example, in the expression f() .OR. g() function g need not be executed if function f returns true (.T.). CA-Clipper performs shortcutting on all logical operators (.OR. .AND. .NOT.).

Side Effect

(modular programming) An unexpected effect of executing a function or program. When a function changes the state of a system in a way that is not explicitly specified by the function's name or calling protocol, the change is called a side effect. Reliance on side effects is contrary to the principles of modular programming.

See Also: Encapsulation Information Hiding Lexical Scoping Modularity

Single Step Mode

(debugger) The mode of execution in which only the line of code highlighted by the Execution Bar is executed, and its output displayed.

Single-dimensional Array

(array) In CA-Clipper, an array whose elements do not contain references to other arrays.

See Also: Array Multidimensional Array Nested Array Subarray Subscript

Skeleton

(command) A wildcard mask used to specify a group of filenames or memory variables. The * is used to specify one or more characters and the ? to specify a single character.

Smart Stringify Result Marker

(preprocessor) A result marker of the form <(id)>. id must correspond to the name of a match marker. A smart stringify result marker specifies that the corresponding source text is to be enclosed in quotes unless the source text was enclosed in parentheses. If the matched source text is a list, each element of the list is individually processed. If no source text was matched, an empty result is produced. The smart stringify result marker is used to implement commands that allow extended expressions (a part of a command that may be either an unquoted literal or a character expression).

Soft Carriage Return

(general) A carriage return that is introduced into text usually in order to implement some sort of wrap operation, as opposed to a Hard Carriage Return that was specifically entered into the text when it was created.

Sort Order

(array, database) Describes the various ways database files and arrays are ordered.

. Ascending

  Causes the order of data in a sort to be from lowest value to highest value.

. Descending

  Causes the order of data in a sort to be from highest value to lowest value.

. Chronological

  Causes data in a sort to be ordered based on a date value, from earliest to most recent.

. ASCII

  Causes data in a sort to be ordered according to the ASCII Code values of the data to be sorted.

. Dictionary

  The data in a sort is ordered in the way it would appear if the items sorted were entries in a dictionary of the English language.

. Collating Sequence

  Data in a sort will be placed in sequence following the order of characters in the IBM Extended Character Set.

. Natural

  The order in which data was entered into the database.

Source Code

(procedure and function) The textual representation of a program or procedure.

See Also: Object File Program File

Source File

```
(configuration)
```

See Also: Header File Program File

Source Text

(preprocessor) Text from a source file, processed by the preprocessor. Source text is examined to see if it matches a previously specified match pattern. If so, the corresponding result pattern is substituted for the matching source text.

Spooler

```
(network)
```

See Also: Print Spooler

Stack

```
(general) A data structure of variable length whose elements are added
and retrieved from the same end.  A stack is often described as first
in, last out.
```

See Also: Queue

Standard Color

(user interface) The color pair definition that is used by all output options (such as SAY and ?), with the exception of GETs and PROMPTs, that use the enhanced color pair.

See Also: Enhanced Color

Statement

(language) In CA-Clipper, the basic unit of source code. A statement is normally a single line of text. Multiple statements can be placed on the same line by separating them with semicolons. A statement may be continued to another line by placing a semicolon at the end of the line to be continued. If the text of a statement matches a command definition (defined with a preprocessor directive), it is translated into the form specified by the command definition.

See Also: Command

Static Overlay

(linker) A section of the program that is not always resident in RAM, and shares memory with other sections. The section that is currently in use is loaded into memory, allowing a larger program to execute in less available RAM.

Static Variable

(variable) A variable that exists and retains its value for the duration of execution. Static variables are lexically scoped; they are only accessible within the procedure that declares them, unless they are declared as filewide, in which case they are accessible to any procedure in the source file that contains the declaration.

See Also: Dynamic Scoping Lexical Scoping Local Variable

STD.CH

(preprocessor) The standard header file containing definitions for all CA-Clipper commands.

Storage Class

(variable) Defines the two characteristics of variables: lifetime and visibility.

See Also: Lifetime Scope Visibility

String

(data type) Generically, a value of type character. In source code, a series of characters enclosed in single or double quotes.

See Also: Character

Stringify

(preprocessor) To change source text into a literal character string by surrounding the text with quotes.

Stub

(error handling) A procedure used for debugging purposes that only simulates the intended actions of the real procedure. It may display an indicating message, return a constant value, or do nothing.

Subarray

(array) In CA-Clipper, an array that is referred to by an element of another array.

See Also: Array Multidimensional Array Nested Array Subscript

`Subdirectory`

`(file)`

See Also: Directory

Subscript

(array) A numeric value used to designate a particular element of an array. Applying a subscript to an array is called subscripting the array. In CA-Clipper programs, subscripting is specified by enclosing a numeric expression in square ([ ]) brackets after the name of a program variable. The variable is then said to be subscripted.

See Also: Array Array Reference Nested Array Subarray

Substring

(data type) A string within a string, usually to be specified as an argument of a function or command.

Swapfile

(linker) Also known as the workfile, used by .RTLink to swap data and code in and out of memory during the linking process.

Symbol

(linker) An assigned name for a value representing a constant or the address of code or data. There are four types of symbols used by the linker defined as follows:

- Absolute symbol: a constant

- Relative symbol: address of code or data

- Public symbol: accessed by modules other than the module in which they are defined. Public symbols are used to share procedures and variables between modules. As such, the relative address of a public symbol is assigned by the compiler during compilation.

- External symbol: a public symbol not defined in the current module. Generally, these are references into CLIPPER.LIB or EXTEND.LIB, but the compiler generates them whenever there is a procedure or user-defined function referenced but not compiled into the current module.

Syntax

(language) The rules that dictate the form of statements or commands as defined by the implementors of the language. Also, a complete description of the forms that a statement or command can take.

Table

(database) A DBMS term defining a collection of column definitions and row values. In CA-Clipper, it is represented and referred to as a database file.

Template

(user interface) A mask that specifies the format in which data should be displayed. For example, you might want to store phone numbers as "9999999999" to save space, but use a template to display the number to the user as "(999) 999-9999."

Text File

```
(file) A file consisting entirely of ASCII characters.  Each line is
separated by a carriage return/linefeed pair (CHR(13) + CHR(10)) and the
file is terminated with a end of file mark (CHR(26)).
```

See Also: Delimited File Program File SDF File

Text Replacement

(preprocessor) The process of removing portions of input text and substituting different text in its place.

Toggle

(command) As a verb, to choose between an on or off state. As a noun, a value or setting that can be either on or off. A toggle is often represented using a logical value, with true (.T.) representing on, and false (.F.) representing off.

Token

(preprocessor) An elemental sequence of characters having a collective meaning. The preprocessor groups characters into tokens as it reads the input text stream. Tokens include identifiers, keywords, constants, and operators. White space, and certain special characters, serve to mark the end of a token to the preprocessor.

Trace Mode

(debugger) A mode of execution similar to Single Step Mode, the difference being that Trace Mode traces over function and procedure calls.

Tracepoint

(debugger) A variable or expression whose value is displayed in the Watch Window, and which causes an application to pause whenever that value changes.

Translation Directive

(preprocessor) A preprocessor instruction containing a translation rule. The two translation directives are #command and #translate.

Translation Rule

(preprocessor) The portion of a translation directive containing a match pattern followed by the special symbol (=>) followed by a result pattern.

Truncate

(expression) To remove insignificant information from the end of an item of data. With numerics, to ignore any part of the number that falls outside of the specified precision.

Tuple

(database) A formal DBMS term that refers to a row in a table or a record in a database file. In DIF files, tuple also refers to the equivalent of a CA-Clipper record.

See Also: Database Field Record

Two-dimensional Array

(array) An array that has two dimensions. In CA-Clipper, an array whose elements contain references to other arrays, all of which have the same length and do not refer to other arrays.

See Also: Array Array Reference Nested Array Subscript

Typeahead Buffer

(user interface)

See Also: Keyboard Buffer

Unary Operator

(expression) An operator that operates on a single operand. For example, the .NOT. operator.

See Also: Binary Operator Operator

Undefine

(preprocessor) To remove an identifier from the preprocessor's list of defined identifiers via the #undefine directive.

Undefined Symbol

(linker) An unresolved symbol) that was never declared public by a module, but which is referenced by another module. After the public symbol definition is encountered, the symbol becomes defined (resolved). When a symbol is referenced, but not defined, it is said to be undefined.

Unselected Color

(user interface) The color pair definition used to display all but the current GET or the GET that has input focus. If this color setting is specified, the current GET is displayed using the current enhanced color.

See Also: Enhanced Color

Update

(database) The process of changing the value of fields in one or more records. Database fields are updated by various commands and the assignment operator.

User Function

(user interface) A user-defined function called by ACHOICE(), DBEDIT(), or MEMOEDIT() to handle key exceptions. A user function is supplied to one of these functions by passing a parameter consisting of a string containing the function's name.

User Interface

(user interface) The way a program interacts with its user (i.e., menu operation and selection, data input methods, etc.)

User-defined Function

(procedure and function)

See Also: Function

Variable

```
(variable) An area of memory that contains a stored value.  Also, the
source code identifier that names a variable.
```

See Also: Local Variable Private Variable Static Variable

Vector

(database) In a DIF file, vector refers to the equivalent of a CA-Clipper field.

See Also: Database Field Record Tuple

Verb

(command) The first word of a command that describes the action to perform.

See Also: Command

View

(database) A DBMS term that defines a virtual table. A virtual table does not actually exist but is derived from existing tables and maintained as a definition. The definition in turn is maintained in a separate file or as an entry in a system dictionary file. In CA-Clipper, views are supported only by DBU.EXE and are maintained in (.vew) files.

See Also: Database Field Record

View Sets Window

(debugger) The window in which CA-Clipper status settings can be inspected.

View Workareas Window

(debugger) The window in which work area information is displayed.

Visibility

```
(variable) The set of conditions under which a variable is accessible by
name.  A variable's visibility depends on its storage class.
```

See Also: Dynamic Scoping Lexical Scoping

Volume

(file) A unit of disk storage uniquely identified by a label and of fixed size. A hard disk can be partitioned into one or more volumes by an operation system utility. Volumes are subdivided into one or more directories organized in tree structure.

See Also: Directory Disk

Wait State

(user interface) A wait state is any mode that extracts keys from the keyboard except for INKEY(). These modes include ACHOICE(), DBEDIT(), MEMOEDIT(), ACCEPT, INPUT, READ and WAIT.

Watch Window

(debugger) The window in which Watchpoints and Tracepoints are displayed.

Watchpoint

(debugger) A variable or expression whose value is displayed in the Watch Window and updated as an application executes.

Wild Match Marker

(preprocessor) A match marker indicating a position that will successfully match any source text. A wild match marker matches all source text from the current position to the end of the source line. A wild match marker has the form <*id*>. id associates a name with the match marker. The name can be used in a result marker to specify how the matching source text is to be handled.

Window

(user interface) A rectangular screen region used for display. A window may be the same size or smaller than the physical screen. Attempting to display information that extends beyond the specified boundaries of the window clips the output at the window edge.

Word

(preprocessor) A series of characters in a match pattern or result pattern. Source text matches a word in a match pattern if the text is identical to the word or is an acceptable abbreviation of it. A word that appears in a result pattern is copied unmodified into the result text.

Word Wrapping

(user interface) The process of continuing the current text on the next line of a display when a boundary is reached and breaking the text on a word boundary.

Work Area

(database) The basic containment area of a database file and its associated indexes. Work areas can be referred to by alias name, number, or a letter designator.

See Also: Alias

`Workfile`

: `(linker)`

See Also: Swapfile

Workstation

(network) A personal computer connected to a network used to run applications and front end processes.

See Also: File Server

Obsolete Language Items

In the Reference guide, the asterisk symbol is used to indicate items in the language that are obsolete or that exist for compatibility with previous releases of CA-Clipper. As you begin to incorporate new product features into your existing applications, it would be advisable to review your code to determine any obsolete items that you might be using. This table lists each obsolete language item with its recommended replacement and should be helpful to you in updating your programs.

Warning: In the future, this table will be the only documentation for obsolete items because they will be removed from the Reference guide. Furthermore, obsolete items are not in keeping with the current CA-Clipper programming philosophy, and we strongly discourage their use as they may not be supported in future releases of CA-Clipper.

Obsolete Language Items

Obsolete	**Recommended**
ADIR() function	DIRECTORY() function
AFIELDS() function	DBSTRUCT() function
CALL command	Extend API
CANCEL command	QUIT command
CLEAR ALL command	CLOSE or RELEASE command
DBEDIT() function	TBrowse class
DBF() function	ALIAS() function
DECLARE statement	PRIVATE statement
DIR command	DIRECTORY() function
DO statement	Use function-calling syntax
EXTERNAL statement	REQUEST statement
FIND command	SEEK command
FKLABEL() function	Use constant value Fn where n ranges from one to 40
FKMAX() function	Use constant value of 40
MOD() function	Modulus operator (%)
NOTE command	C-style comment indicators, /*...*/ and //
READKEY() function	LASTKEY() function
RECCOUNT() function	LASTREC() function
RESTORE SCREEN command	RESTSCREEN() function
SAVE SCREEN command	SAVESCREEN() function
SET COLOR command	SETCOLOR() function
SET EXACT command	Not recommended
SET EXCLUSIVE command	USE command with EXCLUSIVE and SHARED clauses
SET FORMAT command	Not recommended
SET PROCEDURE command	Compiler script files (.clp)
SET UNIQUE command	INDEX command with UNIQUE clause
STORE command	Inline assignment operator (:=)
TEXT command	? or @...SAY command
WAIT command	@...GET...READ command or INKEY() function
WORD() function	Extend API

Character Data Manipulations

Reference Item	Description
ALLTRIM()	Remove leading and trailing spaces from character string
ASC()	Convert a character to its ASCII value
AT()	Return the position of a substring within a string
CHR()	Convert an ASCII code to a character value
CTOD()	Convert a date string to a date value
DTOC()	Convert a date value to a character string
EMPTY()	Determine if the result of an expression is empty
HARDCR()	Replace all soft CRs with hard CRs
ISALPHA()	Determine if the leftmost character is alphabetic
ISDIGIT()	Determine if the leftmost character is a digit
ISLOWER()	Determine if the leftmost character is a lower case letter
ISUPPER()	Determine if the leftmost character is upper case
LEFT()	Extract a substring beginning with the first character
LEN()	Return the length of a character string or array size
LOWER()	Convert uppercase characters to lowercase
LTRIM()	Remove leading spaces from a character string
PAD()	Pad character, date or numeric values with a fill character
RAT()	Return the position of the last occurrence of a substring
REPLICATE()	Return a string repeated a specified number of times
RIGHT()	Return a substring beginning with rightmost character
RTRIM()	Remove trailing spaces from a character string
SET EXACT	Toggle exact matches for character strings
SOUNDEX()	Convert a character string to soundex form
SPACE()	Return a string of spaces
STR()	Convert a numeric expression to a character string
STRTRAN()	Search and replace characters within a character string
STUFF()	Delete and insert characters in a string
SUBSTR()	Extract a substring from a character string
TRANSFORM()	Convert any value into a formatted character string
TYPE()	Determine the type of an expression
UPPER()	Convert lowercase characters to uppercase
VAL()	Convert a character number to numeric type
VALTYPE()	Determine the data type returned by an expression

Memo Field Manipulation

Reference Item	Description
HARDCR()	Replace all soft CRs with hard CRs
MEMOEDIT()	Display or edit character strings and memo fields
MEMOLINE()	Extract a line of text from character string or memo field
MEMOREAD()	Return the contents of a disk file as a character string
MEMOTRAN()	Replace carriage return/line feeds in character strings
MEMOWRIT()	Write a character string or memo field to a disk file
MLCOUNT()	Count the lines in a character string or memo field
MLCTOPOS()	Return byte position based on line and column position
MLPOS()	Determine the position of a line in a memo field
MPOSTOLC()	Return line and column position based on byte position
READINSERT()	Toggle the current insert mode for MEMOEDIT()
SET SCOREBOARD	Toggle the message display from MEMOEDIT()
STRTRAN()	Search and replace characters within a memo field
TYPE()	Determine the type of an expression
VALTYPE()	Determine the data type returned by an expression

Numeric Data Manipulation

Reference Item	Description
ABS()	Return the absolute value of a numeric expression
BIN2I()	Convert a 16-bit signed integer to a numeric value
BIN2L()	Convert a 32-bit signed integer to a numeric value
BIN2W()	Convert a 16-bit unsigned integer to a numeric value
CHR()	Convert an ASCII code to a character value
EMPTY()	Determine if the result of an expression is empty
EXP()	Calculate e**x
INT()	Convert a numeric value to an integer
I2BIN()	Convert a CA-Clipper numeric to a 16-bit binary integer
LOG()	Calculate the natural logarithm of a numeric value
L2BIN()	Convert a numeric value to a 32-bit binary integer
MAX()	Return the larger of two numeric or date values
MIN()	Return the smaller of two numeric or date values
MOD()	Return dBASE III PLUS modulus of two numbers
ROUND()	Return a value rounded to a specified number of digits
SQRT()	Return the square root of a positive number
STR()	Convert a numeric expression to a character string
TRANSFORM()	Convert any value into a formatted character string
TYPE()	Determine the type of an expression
VAL()	Convert a character number to a numeric type
VALTYPE()	Determine the data type returned by an expression
WORD()	Convert CALL command numeric parameters from double to int

BREAK()
Branch out of a BEGIN SEQUENCE...END construct

Syntax

BREAK(<exp>) --> NIL

Arguments

<exp> is the value passed to the RECOVER clause, if any. Note that <exp> is not optional. NIL may be specified if there is no break value.

Returns

BREAK() always returns NIL.

Description

The BREAK() function is identical in functionality to the BREAK statement. The function must be executed during a SEQUENCE.

Examples

. This example illustrates exiting a SEQUENCE from a code block:

```
bSave := ERRORBLOCK( {|x| BREAK(x)} )

BEGIN SEQUENCE
   .
   .
   .
RECOVER USING objError
   .
   .
   .
END

ERRORBLOCK(bSave)
```

Files: Library is CLIPPER.LIB.

See Also: BEGIN SEQUENCE

Date and Time Data Manipulation

Reference Item	Description
CDOW()	Convert a date value to a character day of the week
CMONTH()	Convert a date to a character month name
CTOD()	Convert a date string to a date value
DATE()	Return the system date as a date value
DAY()	Return the day of the month as a numeric value
DOW()	Convert a date value to a numeric day of the week
DTOC()	Convert a date value to a character string
DTOS()	Convert a date value to a string formatted as yyyymmdd
EMPTY()	Determine if the result of an expression is empty
MAX()	Return the larger of two numeric or date values
MIN()	Return the smaller of two numeric or date values
MONTH()	Convert a date value to the number of the month
SECONDS()	Return the number of seconds elapsed since midnight
TIME()	Return the system time
TYPE()	Determine the type of an expression
VALTYPE()	Determine the data type returned by an expression
YEAR()	Convert a date value to the year as a numeric value

Array Manipulation

Reference Item	Description
AADD()	Add a new element to the end of an array
ACHOICE()	Execute a pop-up menu
ACLONE()	Duplicate a nested or multidimensional array
ACOPY()	Copy elements from one array to another
ADEL()	Delete an array element
ADIR()	Fill a series of arrays with directory information
AEVAL()	Execute a code block for each element in an array
AFIELDS()	Fill arrays with the structure of the current database file
AFILL()	Fill an array with a specified value
AINS()	Insert a NIL element into an array
ARRAY()	Create an uninitialized array of specified length
ASCAN()	Scan an array for a value or until a block returns (.T.)
ASIZE()	Grow or shrink an array
ASORT()	Sort an array
ATAIL()	Return the highest numbered element of an array
DBSTRUCT()	Create an array containing the structure of a database file
DECLARE	Create and initialize private memory variables and arrays
DIRECTORY()	Create an array of directory and file information
EMPTY()	Determine if the result of an expression is empty
LEN()	Return the length of a character string or array size
TYPE()	Determine the type of an expression
VALTYPE()	Determine the data type returned by an expression

Code Block Manipulation

Reference Item	Description
AEVAL()	Execute a code block for each element in an array
ASCAN()	Scan an array for a value or until a block returns true
DBEVAL()	Evaluate a code block for each record
ERRORBLOCK()	Post a code block to execute when a runtime error occurs
EVAL()	Evaluate a code block
FIELDBLOCK()	Return a set-get code block for a field variable
FIELDWBLOCK()	Return a set-get block for a field in a given workarea
MEMVARBLOCK()	Return a set-get code block for a given memory variable
SETKEY()	Assign an action block to a key
TYPE()	Determine the type of an expression
VALTYPE()	Determine the data type returned by an expression

Databases

Reference Item	Description
AFIELDS()	Fill arrays with the structure of the current database file
ALIAS()	Return a specified work area alias
APPEND BLANK	Add a new record to current database file
APPEND FROM	Import records from a (.dbf) or ASCII file
AVERAGE	Average numeric expressions in the current work area
BOF()	Determine when beginning of file is encountered
CLEAR ALL	Close files and release public and private variables
CLOSE	Close a specific set of files
COMMIT	Perform a solid-disk write for all active work areas
CONTINUE	Resume a pending LOCATE
COPY STRUCTURE	Copy the current (.dbf) structure to a new (.dbf) file
COPY STRUCTURE EXTENDED	Copy field definitions to a (.dbf) file
COPY TO	Export records to a new (.dbf) or ASCII file
COUNT	Tally records to a variable
CREATE	Create an empty structure extended (.dbf) file
CREATE FROM	Create a new (.dbf) file from a structure extended file
DBAPPEND()	Add a new record
DBCLEARFIL()	Clear a filter condition
DBCLOSEALL()	Close all occupied workareas
DBCLOSEAREA()	Close a workarea
DBCOMMIT()	Flush pending updates
DBCOMMITALL()	Flush pending updates in all workareas
DBCREATE()	Create a database file from a database structure array
DBDELETE()	Mark a record for deletion
DBEVAL()	Evaluate a code block for each record
DBF()	Return current alias name
DBFILTER()	Return the current filter expression as a character string
DBGOBOTTOM()	Move to the last logical record
DBGOTO()	Move to a particular record
DBGOTOP()	Move to the first logical record
DBRECALL()	Reinstate a record marked for deletion
DBSEEK()	Search for a key value
DBSELECTAR()	Change the current workarea
DBSETDRIVER()	Set the default database driver
DBSETFILTER()	Set a filter condition
DBSKIP()	Move relative to the current record
DBSTRUCT()	Create an array containing the structure of a database file
DBUSEAREA()	Use a database file in a workarea
DELETE	Mark records for deletion
DELETED()	Return the deleted status of the current record
EOF()	Determine when end of file is encountered
FCOUNT()	Return the number of fields in the current (.dbf) file
FIELD	Declare database field names
FIELDBLOCK()	Return a set-get code block for a field variable
FIELDGET()	Retrieve the value for a field variable
FIELDNAME()	Return a field name from the current (.dbf) file
FIELDPOS()	Return the position of a field in a workarea
FIELDPUT()	Set the value of a field variable
FIELDWBLOCK()	Return a set-get block for a field in a given workarea
FIND	Search an index for a specified key value
FOUND()	Determine if the previous search operation succeeded
GO	Move the record pointer to a specific record
HEADER()	Return the current database file header length
JOIN	Create a new database file by merging record/fields
LASTREC()	Determine the number of records in the current (.dbf) file
LOCATE	Search sequentially for a record matching a condition
LUPDATE()	Return the last modification date of a (.dbf) file
PACK	Remove deleted records from a database file
RECALL	Restore records marked for deletion
RECCOUNT()	Determine the number of records in the current (.dbf) file
RECNO()	Return the current record number of a work area
RECSIZE()	Determine the record length of a database (.dbf) file

```
 REPLACE       | Assign new values to field variables
 SEEK          | Search an index for a specified key value
 SELECT        | Change the current work area
 SELECT()      | Determine the work area number of a specified alias
 SET DELETED   | Toggle filtering of deleted records
 SET EXCLUSIVE | Establish shared or exclusive USE of database files
 SET FILTER    | Hide records not meeting a condition
 SET PATH      | Specify the CA-Clipper search path for opening files
 SET SOFTSEEK  | Toggle relative SEEKing
 SET UNIQUE    | Toggle the inclusion of nonunique keys into an index
 SKIP          | Move the record pointer to a new position
 SORT          | Copy to a (.dbf) file in sorted order
 SUM           | Sum numeric expressions to variables
 TOTAL         | Summarize records by key value to a (.dbf) file
 UPDATE        | Update current database file from another database file
 USE           | Open an existing (.dbf) and its associated files
 USED()        | Determine if a database file is in USE
 ZAP           | Remove all records from the current database file
-------------------------------------------------------------------------------
```

Indexes

Reference Item	Description
DBCLEARIND()	Close indexes
DBCREATEIND()	Create an index file
DBREINDEX()	Recreate all active indexes
DBSEEK()	Search for a key value
DBSETINDEX()	Open an index
DBSETORDER()	Set the controlling order
DESCEND()	Create a descending index key value
FOUND()	Determine if the previous search operation succeeded
INDEX	Create an index file
INDEXEXT()	Return the default index extension
INDEXKEY()	Return the key expression of a specified index
INDEXORD()	Return the order position of the controlling index
REINDEX	Rebuild open indexes in the current workarea
SET INDEX	Open index file(s) in the current work area
SET ORDER	Set a new controlling index
SET UNIQUE	Toggle the inclusion of nonunique keys into an index

Relations

Reference Item	Description
DBCLEAREL()	Clear active relations
DBRELATION()	Return the linking expression of a specified relation
DBRSELECT()	Return the target workarea number of a relation
DBSETRELAT()	Relate two workareas
SET RELATION	Relate two work areas by a key value or record number

Program Execution

Reference Item	Description
BEGIN SEQUENCE	Define a sequence of statements for a BREAK
BREAK()	Branch out of a BEGIN SEQUENCE...END construct
CALL	Execute a C or Assembler procedure
CANCEL	Terminate program processing
DO	Call a procedure
DO CASE	Execute one of several alternative blocks of statements
DO WHILE	Execute a loop while a condition is true (.T.)
EXTERNAL	Declare a list of procedure or user-defined function names
FOR	Execute a block of statements a specified number of times
FUNCTION	Declare a user-defined function name and formal parameters
IF	Execute one of several alternative blocks of statements
IF()	Return the result of an expression based on a condition
NOTE	Place a single-line comment in a program file
PARAMETERS	Create private parameter variables
PCOUNT()	Determine the position of the last actual parameter passed
QUIT	Terminate program processing
RETURN	Terminate a procedure, user-defined function or program
RUN	Execute a DOS command or program
SET KEY	Assign a procedure invocation to a key
SET PROCEDURE	Compile procedures/functions into the current .OBJ file
SETKEY()	Assign an action block to a key
SETCANCEL()	Toggle Alt-C as a program termination key
WAIT	Suspend program processing until a key is pressed
WORD()	Convert CALL command numeric parameters from double to int

Memory Variable Handling

Reference Item	Description
ACCEPT	Place keyboard input into a memory variable
CLEAR ALL	Close files and release public and private variables
CLEAR MEMORY	Release all public and private variables
DECLARE	Create and initialize private memory variables and arrays
INPUT	Enter the result of an expression into a variable
LOCAL	Declare and initialize local variables and arrays
MEMVAR	Declare private and public variable names
MEMVARBLOCK()	Return a set-get code block for a given memory variable
PARAMETERS	Create private parameter variables
PRIVATE	Create and initialize private memory variables and arrays
PUBLIC	Create and initialize public memory variables and arrays
STATIC	Declare and initialize static variables and arrays
RELEASE	Delete public and private memory variables
RESTORE	Retrieve memory variables from a memory (.mem) file
SAVE	Save variables to a memory (.mem) file
STORE	Assign a value to one or more variables

Debugging and Error Handling

Reference Item	Description
ALTD()	Invoke the CA-Clipper Debugger
BREAK()	Branch out of a BEGIN SEQUENCE...END construct
DOSERROR()	Return the DOS error number
ERRORBLOCK()	Post a code block to execute when a runtime error occurs
ERRORLEVEL()	Set the CA-Clipper return code
NETERR()	Determine if a network command has failed
OUTERR()	Write a list of values to the standard error device
PROCLINE()	Return the source line number of an activation
PROCNAME()	Return the name of a procedure or user-defined function

BROWSE()*
Browse records within a window

Syntax

BROWSE([<nTop>], [<nLeft>],
[<nBottom>], [<nRight>]) --> lSuccess

Arguments

<nTop>, **<nLeft>**, **<nBottom>**, and **<nRight>** define the window coordinates. If not specified, the default window coordinates are 1, 0 to MAXROW(), and MAXCOL().

Returns

BROWSE() returns false (.F.) if there is no database file in use; otherwise, it returns true (.T.).

Description

BROWSE() is a user interface function that invokes a general purpose table-oriented browser and editor for records in the current work area. For a list of the navigation keys which are used by BROWSE(), refer to the DBEDIT() function.

Notes

. **Status line:** BROWSE() supports a status line in the upper right corner of the browse window indicating one of the following:

BROWSE() Status Line Messages

Message	Meaning
<new>	Append mode
<bof>	Top of file
<delete>	Current record is deleted
Record	Record number display

. BROWSE() has the following three modes:

- **Browsing:** This is the default mode of BROWSE(). Pressing any DBEDIT() navigation key moves the highlight to a new column or row.

- **Field edit:** Pressing Return on any field enters field edit using a GET. Pressing Return terminates the edit mode, saving the changes. Esc terminates without saving changes. Since the field edit mode uses GET, all navigation and editing keys are READ keys.

- **Append:** GOing BOTTOM with Ctrl-PgDn and then pressing Dnarrow enters append mode with the indicating message "<new>" on the status line. A new blank record is then inserted. Pressing Uparrow terminates the append mode, saving the new record if data has been entered. If no data has been entered, the new record is not saved.

Examples

. This is an example of browsing a file:

```
USE File1 NEW
BROWSE()
```

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\BROWSE.PRG.

See Also: DBEDIT()*

Low-level File Handling

Reference Item	Description
FCLOSE()	Close an open binary file and write DOS buffers to disk
FCREATE()	Create and/or truncate a binary file to zero length
FERASE()	Delete a file from disk
FERROR()	Test for errors after a binary file operation
FOPEN()	Open a binary file
FREAD()	Read characters from a binary file into a buffer variable
FREADSTR()	Read characters from a binary file
FSEEK()	Set a binary file pointer to a new position
FWRITE()	Write to an open binary file
HEADER()	Return the current database file header length

Data Input and Output

Reference Item	Description
?/??	Display one or more values to the console
@...BOX	Draw a box on the screen
@...CLEAR	Clear a rectangular region of the screen
@...GET	Create a new Get object and display it on the screen
@...PROMPT	Paint a menu item and define a message
@...SAY	Display data at a specified screen or printer row and column
@...TO	Draw a single or double line box
ACHOICE()	Execute a pop-up menu
ALERT()	Display a simple modal dialog box
BROWSE()	Browse records within a window
CLEAR GETS	Release Get objects from the current GetList array
CLEAR SCREEN	Clear the screen and home the cursor
COL()	Return the screen cursor column position
DBEDIT()	Browse records in a table format
DEVOUT()	Write a value to the current device
DISPBEGIN()	Begin buffering screen output
DISPBOX()	Display a box on the screen
DISPEND()	Display buffered screen updates
DISPLAY	Display records to the console
DISPOUT()	Write a value to the display
GETACTIVE()	Return the currently active Get object
LABEL FORM	Display labels to the console
LIST	List records to the console
MENU TO	Execute a lightbar menu for defined PROMPTs
OUTSTD()	Write a list of values to the standard output device
QOUT()	Display a list of expressions to the console
READ	Activate full-screen editing mode using Get objects
READEXIT()	Toggle Uparrow and Downarrow as READ exit keys
READINSERT()	Toggle the current insert mode for READ and MEMOEDIT()
READKEY()	Determine what key was used to terminate a READ
READMODAL()	Activate a full-screen editing mode for a GetList
READVAR()	Return the current GET/MENU variable name
REPORT FORM	Display a report to the console
RESTORE SCREEN	Display a saved screen
RESTSCREEN()	Display a saved screen region to a specified location
ROW()	Return the screen row position of the cursor
SAVE SCREEN	Save current screen to a buffer or variable
SAVESCREEN()	Save a screen region for later display
SCROLL()	Scroll a screen region up or down
SET CENTURY	Modify the date format to include or omit century digits
SET COLOR	Define screen colors
SET CONFIRM	Toggle required exit key to terminate GETs
SET CONSOLE	Toggle console display to the screen
SET CURSOR	Toggle the screen cursor on or off
SET DATE	Set the date format for input and display
SET DECIMALS	Set the number of decimal places displayed
SET DELIMITERS	Toggle or define GET delimiters
SET DEVICE	Direct @...SAYs to the screen or printer
SET EPOCH	Control the interpretation of dates with no century digits
SET ESCAPE	Toggle Esc as a READ exit key
SET FIXED	Toggle fixing of the number of decimal digits displayed
SET FORMAT	Activate a format when READ is executed
SET INTENSITY	Toggle enhanced display of GETs and PROMPTs
SET MESSAGE	Set the @...PROMPT message line row
SET SCOREBOARD	Toggle the message display from READ or MEMOEDIT()
SET WRAP	Toggle wrapping of the highlights in MENUs
SETBLINK()	Toggle asterisk (*) interpretation in SET COLOR
SETCOLOR()	Return the current colors and optionally set new colors
SETCURSOR()	Set the cursor shape
SETMODE()	Change display mode to specified number of rows and columns
SETPOS()	Move the cursor to a new position
TEXT	Display a literal block of text
TRANSFORM()	Convert any value into a formatted character string

Printing

Reference Item	Description
?/??	Display one or more values to the console
@...SAY	Display data at a specified screen or printer row and column
DEVPOS()	Move the printhead to a new position
DISPLAY	Display records to the console
EJECT	Advance the print head to top of form
ISPRINTER()	Determine whether the LPT1 is ready
LABEL FORM	Display labels to the console
LIST	List records to the console
OUTSTD()	Write a list of values to the standard output device
PCOL()	Return the current column position of the printhead
PROW()	Return the current row position of the printhead
QOUT()	Display a list of expressions to the console
REPORT FORM	Display a report to the console
SET MARGIN	Set the page offset for all printed pages
SET PRINTER	Toggle echo of console output to the printer
SETPRC()	Set PROW() and PCOL() values
TYPE	Display the contents of a text file

Keyboard Entry

Reference Item	**Description**
CLEAR TYPEAHEAD	Empty the keyboard buffer
INKEY()	Extract a character from the keyboard buffer
KEYBOARD	Stuff a string into the keyboard buffer
LASTKEY()	Return the INKEY() value of the last key in the buffer
NEXTKEY()	Read the pending key in the keyboard buffer
SET TYPEAHEAD	Set the size of the keyboard buffer

Environment

Reference Item	Description
CURDIR()	Return the current DOS directory
DISKSPACE()	Return the space available on a specified disk
FKLABEL()	Return function key name
FKMAX()	Return number of function keys as a constant
GETENV()	Retrieve the contents of a DOS environment variable
ISCOLOR()	Determine if the current computer has color compatibility
MAXCOL()	Determine the maximum visible screen column
MAXROW()	Determine the maximum visible screen row
MEMORY()	Determine the amount of available free pool memory
NOSNOW()	Toggle snow position
OS()	Return the operating system name
SET()	Inspect or change a global setting
SET BELL	Toggle automatic sounding of the bell
SET COLOR	Define screen colors
SET DEFAULT	Set the CA-Clipper default drive and directory
SET FUNCTION	Assign a character string to a function key
SET TYPEAHEAD	Set the size of the keyboard buffer
TONE()	Sound a speaker tone for a specified frequency and duration
VERSION()	Returns CA-Clipper version

File Management

Reference Item	Description
ADIR()	Fill a series of arrays with directory information
CLOSE	Close a specific set of files
COPY FILE	Copy a file to a new file
CURDIR()	Return the current DOS directory
DELETE FILE	Remove a file from disk
DIR	Display a listing of files from a specified path
DIRECTORY()	Create an array of directory and file information
DISKSPACE()	Return the space available on a specified disk
ERASE	Remove a file from disk
FILE()	Determine if files exist in the default directory or path
FRENAME()	Change the name of a file
RENAME	Change the name of a file
SET ALTERNATE	Echo console output to a text file
SET DEFAULT	Set the CA-Clipper default drive and directory
SET PATH	Specify the CA-Clipper search path for opening files
TYPE	Display the contents of a text file

Networking

Reference Item	Description
DBUNLOCK()	Release all locks for the current workarea
DBUNLOCKALL()	Release all locks for all workareas
FLOCK()	Lock an open and shared database file
NETERR()	Determine if a network command has failed
NETNAME()	Return the current workstation identification
RLOCK()	Lock the current record in the active work area
SET EXCLUSIVE	Establish shared or exclusive USE of database files
UNLOCK	Release file record locks set by the current user

Preprocessor Directives

Directive	Description
#command/ #translate	Specify a user-defined command or translation directive
#define	Define a manifest constant or pseudofunction
#error	Generate a compiler error and display a message
#ifdef	Compile a section of code if an identifier is defined
#ifndef	Compile a section of code if an identifier is undefined
#include	Include a file into the current source file
#stdout	Outputs the literal text to the standard output device
#undef	Remove a #define definition
#xcommand/ #xtranslate	Specify a user-defined command or translation directive

String Operators

Operator	Description
+	Concatenate
-	Concatenate without intervening spaces

Mathematical Operators

Operator	Description
+	Addition or Unary Positive
-	Subtraction or Unary Negative
*	Multiplication
/	Division
%	Modulus
** or ^	Exponentiation

Relational Operators

Operator	Description
<	Less than
>	Greater than
=	Equal
==	Exactly equal for character; equal for other data types
<>, #, or !=	Not equal
<=	Less than or equal
>=	Greater than or equal
$	Is contained in the set or is a subset of

Logical Operators

Operator	Description
.AND.	Logical and
.OR.	Logical or
.NOT. or !	Logical negate

Assignment Operators

Operator	Description
=	Assign
:=	Inline assign
+=	Inline add (or concatenate) and assign
-=	Inline subtract (or concatenate) and assign
*=	Inline multiply and assign
/=	Inline divide and assign
^=	Inline exponentiate and assign
%=	Inline modulus and assign

Increment and Decrement Operators

Operator	Description
++	Prefix or postfix increment
--	Prefix or postfix decrement

Special Operators

Operator	Description
:	Send a message to an object
()	Function or grouping
[]	Array element
{}	Constant array definition
->	Alias identifier
&	Compile and execute
@	Pass by reference

COMPUTER ASSOCIATES INTERNATIONAL, INC.
LICENSE AGREEMENT AND
LIMITED PRODUCT WARRANTY

CAREFUL READ THE FOLLOWING TERMS AND CONDITIONS. YOUR OPENING OF THE PROGRAM PACKET WILL INDICATE YOUR ACCEPTANCE. IF YOU DO NOT AGREE WITH THESE TERMS AND CONDITIONS, YOU SHOULD PROMPTLY RETURN THE COMPLETE PACKAGE AND YOUR MONEY WILL BE REFUNDED.

Computer Associates provides this Program and licenses its use to you. You are responsible for selecting the Program to achieve your intended results and for the installation, use and results obtained from the Program.

THE PROGRAM, INCLUDING ITS CODE, DOCUMENTATION, APPEARANCE, STRUCTURE AND ORGANIZATION, IS A PROPRIETARY PRODUCT OF COMPUTER ASSOCIATES AND IS PROTECTED BY COPYRIGHT AND OTHER LAWS. TITLE TO THE PROGRAM, OR ANY COPY, MODIFICATION OR MERGED PORTION OF THE PROGRAM, SHALL AT ALL TIMES REMAIN WITH COMPUTER ASSOCIATES.

LICENSE--The following restricted rights are granted:

You May:

a. Use the Program only on a single computer or install the Program on a single computer network system for your own individual use. Even though copies of the Program may be provided on disks of different sizes, you may not use both sizes simultaneously on different computers. The Program may be transferred to and used on another computer, so long as the Program is de-installed from the original computer, and shall under no circumstance be used on more than one computer at a time.

b. Make one copy of the Program for backup or archival purposes in support of your permitted use of the Program.

c. Transfer the Program together with this License to another person, but only if the other person agrees to accept the terms and conditions of this Agreement. If you transfer the Program and License, you must at the same time either transfer all copies of the Program to the same person or destroy those not transferred. Any such transfer terminates your License, but you agree that you shall continue to keep all Computer Associates materials confidential.

d. If the Program is a language product, reproduce and distribute the runtime modules included in the Program (identified in the accompanying written materials) required during the execution of software products that you design or develop ("your software product") using the Program, so long as: (1) the modules are reproduced unaltered and distributed solely in connection with and as part of your software product; (2) Computer Associates' name, logo or trademarks are not used to market your software product; (3) Computer Associates is indemnified, held harmless and defended by you from and against any claims, lawsuits or causes of action, in equity or in law, including related attorney's fees, that may arise from the distribution or use of your software product; and (4) Computer Associates' copyright notice for the Program is set forth on the label and welcome screen for your software product.

YOU MAY NOT:

a. TRANSFER OR RENT THE PROGRAM OR USE, COPY OR MODIFY THE PROGRAM, IN WHOLE OR IN PART, EXCEPT AS EXPRESSLY PERMITTED IN

THIS AGREEMENT.

b. DECOMPILE, REVERSE ASSEMBLE OR OTHERWISE REVERSE ENGINEER THE PROGRAM.

c. REPRODUCE, DISTRIBUTE OR REVISE THE PROGRAM DOCUMENTATION.

IF YOU DO ANY OF THE FOREGOING, YOUR LICENSE AND THIS AGREEMENT ARE AUTOMATICALLY TERMINATED. SUCH TERMINATION SHALL BE IN ADDITION TO AND NOT IN LIEU OF ANY CRIMINAL, CIVIL OR OTHER REMEDIES AVAILABLE TO COMPUTER ASSOCIATES.

LIMITED WARRANTY

EXCEPT AS SPECIFICALLY STATED IN THIS AGREEMENT, THE PROGRAM IS PROVIDED AS LICENSED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED, TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Computer Associates warrants that the Program will substantially perform the functions or generally conform to the Program's specifications published by Computer Associates and included in this package.

Computer Associates warrants that the disks on which the Program is furnished will be free from defects in materials and workmanship under normal use for a period of 90 days from the date of delivery to you.

Computer Associates does not warrant that the functions contained in the Program will meet your requirements or that the operation of the Program will be entirely error free or appear precisely as described in the Program documentation.

LIMITATION OF REMEDIES AND LIABILITY

To the maximum extent permitted by applicable law, the remedies described below are accepted by you as your only remedies, and shall be available to you only if you or your dealer returns the enclosed registration form to Computer Associates within ten days after delivery of the Program to you.

Computer Associates' entire liability and your exclusive remedies shall be:

a. If the Program does not substantially perform the functions or generally conform to the Program's specifications published by Computer Associates, you may, within 90 days after delivery, write to Computer Associates to report a significant defect. If Computer Associates is unable to correct that defect within 90 days after receiving your report, you may terminate your License and this Agreement by returning the Program with your original receipt and your money will be refunded.

b. If the Program disks are defective, within 90 days of delivery, you may return it with a copy of your receipt and Computer Associates will replace it.

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL COMPUTER ASSOCIATES BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING LOST PROFITS, LOST SAVINGS, OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES, ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM, EVEN IF COMPUTER ASSOCIATES OR A DEALER AUTHORIZED BY COMPUTER ASSOCIATES HAD BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

GENERAL

Computer Associates does not guarantee the performance of the Program for use on a network system because the configuration and type of network can affect such performance.

If any provision of this Agreement is held to be unenforceable, the enforceability of the remaining provisions shall in no way be affected or impaired thereby.

Any questions concerning this Agreement should be referred in writing to Computer Associates International, Inc., One Computer Associates Plaza, Islandia, NY 11788-7000.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT AND, BY OPENING THE PROGRAM PACKET, INDICATE YOUR ACCEPTANCE OF ITS TERMS AND CONDITIONS. YOU ALSO AGREE THAT, UNLESS SPECIFICALLY COVERED BY ANOTHER LICENSE AGREEMENT, IT IS THE COMPLETE AGREEMENT BETWEEN US AND THAT IT SUPERSEDES ANY INFORMATION YOU RECEIVED RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

Error
Provides objects containing information about runtime errors

Description

An Error object is a simple object that contains information pertaining to a runtime error. Error objects have no methods, only exported instance variables. When a runtime error occurs, CA-Clipper creates a new Error object and passes it as an argument to the error handler block specified with the ERRORBLOCK() function. Within the error handler, the Error object can then be queried to determine the nature of the error condition.

Error objects can also be returned to the RECOVER statement of a BEGIN SEQUENCE construct with a BREAK statement. Here, the error object can be queried for local error handling. For more detailed information and examples refer to the Error Handling Strategies chapter in the Programming and Utilities guide.

Examples

. This example demonstrates how a file open operation might be handled in an error handler replicating the default CA-Clipper behavior. When, for example, an attempt to open a database file with a USE command fails, control returns to the statement following the offending command:

```
#include "Error.ch"
#command RETRY      => RETURN (.T.)    // Retry operation
#command RESUME     => RETURN (.F.)    // Default recovery
//

FUNCTION MyError( objError )
   //
   // Handle file open error
   IF objError:GenCode == EG_OPEN .AND.;
      objError:CanDefault .AND.;
      NETERR()
      //
      RESUME
   ENDIF
   .
   . <other error statements>
   .
   RETURN NIL
```

. This example retries an operation within an error handler a specified number of times:

```
#include "Error.ch"
#command RETRY      => RETURN (.T.)    // Retry operation
#command RESUME     => RETURN (.F.)    // Default recovery
//

FUNCTION MyError( objError )
   //
   // Handle printer not ready error
   IF objError:GenCode == EG_PRINT .AND.;
      objError:CanRetry .AND.;
      objError:Tries < 25
      //
      RETRY
   ENDIF
   .
   . <other error statements>
   .
   RETURN NIL
```

. This code fragment returns an error object from an error handler to the RECOVER statement for further processing:

```
LOCAL objLocal, bLastHandler
//
// Save current and set new error handler
bLastHandler := ERRORBLOCK({    |objErr| ;
                  MyHandler(objErr, .T.)})
//
BEGIN SEQUENCE
   .
   . <operation that might fail>
   .
RECOVER USING objLocal
   .
   . <send messages to objLocal and handle the error>
   .
END
//
// Restore previous error handler
ERRORBLOCK( bLastHandler )

FUNCTION MyHandler( objError, lLocalHandler )
   //
   // Handle locally returning the error object
   IF lLocalHandler
      BREAK objError
   ENDIF
   .
   . <other statements to handle the error>
   .
   RETURN NIL


Check the samples on ERRORSYS in the \CLIPPER5\SOURCE\ERRORSYS
subdirectory for working examples on ERRORSYS.
```

Files: Header file is Error.ch, default error handler is in Errorsys.prg.

See Also: BEGIN SEQUENCE DOSERROR() ERRORBLOCK() NETERR()

ErrorNew() --> objError

```
   Returns a new Error object.

   oNewErr := ErrorNew()

   // Assign some instance variables
   oNewErr:CanDefault    := .F.
   oNewErr:CanSubstitute := .F.
   oNewErr:CanRetry      := .F.
   oNewErr:Severity      := ES_CATASTROPHIC
   oNewErr:Description   := ERR_UNKNOWN
   oNewErr:Operation     := ERR_WHOKNOWS
```

Args **(Assignable)**

Contains an array of the arguments supplied to an operator or function when an argument error occurs. For other types of errors, Error:Args contains a NIL value.

```
QOUT( "Args          : " ); QQOUT( oErr:Args )
```

CanDefault **(Assignable)**

Contains a logical value indicating whether the subsystem can perform default error recovery for the error condition. A value of true (.T.) indicates that default recovery is available. Availability of default handling and the actual default recovery strategy depends on the subsystem and the error condition. The minimum action is simply to ignore the error condition.

Default recovery is requested by returning false (.F.) from the error block invoked to handle the error. Note that Error:CanDefault is never true (.T.) if Error:CanSubstitute is true (.T.).

```
oNewErr:CanDefault := .F.
```

CanRetry **(Assignable)**

Contains a logical value indicating whether the subsystem can retry the operation that caused the error condition. A value of true (.T.) indicates that a retry is possible. Retry may or may not be available, depending on the subsystem and the error condition.

Retry is requested by returning true (.T.) from the error block invoked to handle the error. Note that Error:CanRetry never contains true (.T.) if Error:CanSubstitute contains true (.T.).

```
oNewErr:CanRetry := .F.
```

CanSubstitute **(Assignable)**

Contains a logical value indicating whether a new result can be substituted for the operation that produced the error condition. Argument errors and certain other simple errors allow the error handler to substitute a new result value for the failed operation. A value of true (.T.) means that substitution is possible.

The substitution is performed by returning the new result value from the code block invoked to handle the error. Note that Error:CanSubstitute is never true (.T.) if either Error:CanDefault or Error:CanRetry is true (.T.).

```
oNewErr:CanSubstitute := .F.
```

Cargo **(Assignable)**

Contains a value of any data type unused by the Error system. It is provided as a user-definable slot, allowing arbitrary information to be attached to an Error object and retrieved later.

```
oNewErr:Cargo := {|x| MyUDF(x) }
```

Description **(Assignable)**

Contains a character string that describes the error condition. A zero-length string indicates that the subsystem does not provide a printable description for the error. If Error:GenCode is not zero, a printable description is always available.

```
oNewErr:Description := ERR_UNKNOWN
```

FileName **(Assignable)**

Contains a character value representing the name originally used to open the file associated with the error condition. A zero-length string indicates either that the error condition is not associated with a particular file or that the subsystem does not retain FileName information.

```
oNewErr:FileName := "Customer.dbf"
```

GenCode **(Assignable)**

Contains an integer numeric value representing a CA-Clipper generic error code. Generic error codes allow default handling of similar errors from different subsystems. A value of zero indicates that the error condition is specific to the subsystem and does not correspond to any of the generic error codes. The header file, Error.ch, provides a set of manifest constants for generic error codes.

```
If oNewErr:GenCode == EG_ZERODIV
   .
   .
   .

Endif
```

Operation **(Assignable)**

Contains a character string that describes the operation being attempted when the error occurred. For operators and functions, Error:Operation contains the name of the operator or function. For undefined variables or functions, it contains the name of the variable or function. A zero-length string indicates that the subsystem does not provide a printable description of the operation.

```
oNewErr:Operation := ERR_WHOKNOWS
```

OsCode **(Assignable)**

Contains an integer numeric value representing the operating system error code associated with the error condition. A value of zero indicates that the error condition was not caused by an error from the operating system. When Error:OsCode is set to a value other than zero DOSERROR() is updated with the same value.

Error:OsCode properly reflects the DOS extended error code for file errors. This allows proper distinction between errors which result from sharing violations (e.g., opening EXCLUSIVE when another process has already opened the file) and access violations (e.g., opening read/write when the file is marked read-only).

For a list of DOS error codes refer to the Error Messages and Appendices guide.

```
If oNewErr:OsCode == DOS_SHARING_ERROR
   .
   .
   .

Endif
```

Severity **(Assignable)**

Contains a numeric value indicating the severity of the error condition. Four standard values are defined in Error.ch:

Error:Severity Values

Error.ch	Meaning
ES_WHOCARES	The condition does not represent a failure; the error is informational.
ES_WARNING	The condition does not prevent further operations, but may result in a more serious error later.
ES_ERROR	The condition prevents further operations without corrective action of some kind.
ES_CATASTROPHIC	The condition requires immediate termination of the application.

Note that the CA-Clipper runtime support code only generates errors with severities of ES_WARNING or ES_ERROR.

```
oNewErr:Severity := ES_CATASTROPHIC
```

SubCode **(Assignable)**

Contains an integer numeric value representing a subsystem-specific error code. A value of zero indicates that the subsystem does not assign any particular number to the error condition.

```
QOUT( "SubCode       : " ); QQOUT( oNewErr:SubCode )
```

SubSystem **(Assignable)**

Contains a character string representing the name of the subsystem generating the error. For errors with basic CA-Clipper operators and functions, the subsystem name "BASE" is given. For errors generated by a database driver, Error:SubSystem contains the name of the database driver.

```
QOUT( "SubSystem       : " ); QQOUT( oNewErr:SubSystem )
```

Tries **(Assignable)**

Contains an integer numeric value representing the number of times the failed operation has been attempted. When Error:CanRetry is true (.T.), Error:Tries can be used to limit the number of retry attempts. A value of zero indicates that the subsystem does not track the number of times the operation has been tried.

```
If (oNewErr:GenCode == EG_OPEN .AND.;
    oNewErr:OsCode == DOS_SHARING_ERROR .AND.;
    oNewErr:CanRetry .AND. oNewErr:Tries < MAX_NOF_ATTEMPTS)
    .
    .
    .

Endif
```

Get
Provides objects for interactive editing of database fields and variables

---

Description

A Get object is a general purpose mechanism for editing data. It is used in CA-Clipper to implement the @...GET and READ commands. Get objects provide a sophisticated architecture for formatting and editing data, including cursor navigation and data validation. Data validation is performed via user-supplied code blocks, and display formatting can be controlled using standard picture strings.

Normally, a Get object is associated with a particular variable (referred to as the Get variable). The Get object does not directly access this variable; instead, the variable is manipulated by evaluating a supplied code block. When a Get object is created using the standard @...GET command, a code block is automatically created which provides access to the variable named in the command. For a Get object created with the GetNew() function, you must specify an appropriate code block.

A Get object evaluates the supplied code block to retrieve the value of the Get variable. The value is converted to displayable form and copied into a buffer referred to as the editing buffer. You can display the editing buffer on the screen for editing. Programmable behaviors allow the navigation of the editing buffer and subsequent copying of its contents back to the Get variable.

Examples

. This example creates a new Get object, assigns some new attributes to its instance variables, then edits the Get with the READMODAL() function:

```
LOCAL cVar := SPACE(10)
//

// Create a new Get object
objGet := GetNew()
//

// Assign some instance variables
objGet:Row := 10
objGet:Vol := 10
//

// Assign the name of the associated
// variable and the block
objGet:Name := "cVar"
objGet:Block := { |cValue| IF(PCOUNT() > 0,;
                  cVar := cValue, cVar) }
//
objGet:Picture := "@!"
objGet:ColorSpec := "BG+/B, W+/BG"
objGet:PostBlock := { |oGet| !EMPTY(oGet:VarGet()) }
//

// Edit the single Get object
READMODAL({objGet})
```

Files: Source file is Getsys.prg.

See Also: @...GET READ READMODAL() READVAR() SETCOLOR()

GetNew([<nRow>], [<nCol>], [<bBlock>], [<cVarName>],
[<cPicture>], [<cColorSpec>]) --> objGet

Returns a new Get object with the row, col, block, picture, and colorSpec instance variables set from the supplied arguments.

```
oGet := GetNew( 10, 10, {|x| IF(x == NIL, cVar, cVar := x)},;
                "cVar", "@R (999)999-9999", "N/BG, W+/BG" )
```

CDOW()
Convert a date value to a character day of the week

Syntax

CDOW(<dExp>) --> cDayName

Arguments

<dExp> is the date value to convert.

Returns

CDOW() returns the name of the day of the week as a character string. The first letter is uppercase and the rest of the string is lowercase. For a null date value, CDOW() returns a null string ("").

Description

CDOW() is a date conversion function used in formatting date displays for reports, labels, and screens.

Examples

. These examples illustrate CDOW():

```
? DATE()                      // Result: 09/01/90
? CDOW(DATE())                // Result: Friday
? CDOW(DATE() + 7)            // Result: Friday
? CDOW(CTOD("06/12/90"))      // Result: Tuesday
```

Files: Library is CLIPPER.LIB.

See Also: CTOD() DATE() DAY() DOW()

BadDate

Contains a logical value indicating the editing buffer does not represent a valid date. Get:BadDate contains true (.T.) when the Get object is a date type and the date represented by the contents of the editing buffer is invalid. Get:BadDate contains false (.F.) when the date is valid or the Get is not editing a date value.

```
If (oGet:BadDate)
   .
   .
   .

Endif
```

Block **(Assignable)**

Contains a code block that associates the Get object with a variable. The code block takes an optional argument that should assign the value of the argument to the variable. If the argument is omitted, the code block should return the current value of the variable.

If the Get variable is an array element, Get:Block always returns the base of the array. The subscript(s) in the expression are stored internally. Thus, in the case of GETs on array elements, you cannot assign or retrieve the value of the array element by executing Get:Block. Setting and getting may be done on array element GET variables (and simple GET variables as well) by use of the VarGet() and VarPut() methods, defined below.

Note: Use of the VarGet and VarPut messages--instead of directly evaluating the variable block in the get--is the preferred method of accessing the Get variable.

```
oGet       := GetNew()
oGet:Row   := 10
oGet:Col   := 10
oGet:Block := {|x| IF(x == NIL, cVar, cVar := x)}

// or you can use this Get/Set block
oGet:Block := {|x| IF( pcount() > 0, cVar := x, cVar)}
```

Buffer **(Assignable)**

Contains a character value which is the editing Buffer used by the Get object. Get:Buffer is meaningful only when the Get object has input focus. At other times, it contains NIL and any attempts to assign a new value are ignored.

```
oGet:SetFocus()
If (oGet:Buffer) == ""
   .
   .
   .

Endif
```

Cargo **(Assignable)**

Contains a value of any data type unused by the Get system. Get:Cargo is provided as a user-definable slot, allowing arbitrary information to be attached to a Get object and retrieved later.

```
oGet:Cargo := {|x| MyUDF(x) }
```

Changed

Contains a logical value indicating whether the Get:Buffer has changed since the Get received input focus. Get:Changed contains true (.T.) if the buffer has changed; otherwise it contains false (.F.).

```
If (oGet:Changed)
   .
   .
   .

Endif
```

Clear **(Assignable)**

Contains a logical value indicating whether the editing buffer should be cleared before any more values are entered. Get:Clear is set true (.T.) by Get:SetFocus() and Get:Undo() when the Get variable is a numeric type or when Get:picture contains the "@K" picture function. At all other times, it contains false (.F.).

```
If !(oGet:Clear)
   .
   .
   .

Endif
```

Col **(Assignable)**

Contains a numeric value defining the screen column where the Get is displayed.

```
oGet:Col := 10
```

ColorSpec **(Assignable)**

Contains a character string defining the display attributes for the Get object. The string must contain two color specifiers. The first, called the unselected color, determines the color of the Get object when it does not have input focus. The second, called the selected color, determines the color of the Get when it has input focus.

If no colors are specified, Get:ColorSpec is initialized using the current SETCOLOR() colors. The SETCOLOR() unselected and enhanced colors are used as the Get object's unselected and selected colors, respectively. See the SETCOLOR() entry in this chapter for more information on color specifiers.

```
oGet:ColorSpec := "N/BG, W+/BG"
```

decPos

Contains a numeric value indicating the decimal point position within the editing buffer. Get:DecPos is meaningful only when the value being edited is numeric and the Get object has input focus. Otherwise, it contains NIL.

```
oGet:SetFocus()
If (oGet:DecPos == 2)
   .
   .
   .

Endif
```

ExitState **(Assignable)**

Contains a numeric value used in the CA-Clipper version of Getsys.prg to record the means by which a Get object was exited.

Get Exit States

Number	Getexit.ch	Meaning
0	GE_NOEXIT	No exit attempted, prepare Get for editing
1	GE_UP	Go to previous Get
2	GE_DOWN	Go to next Get
3	GE_TOP	Go to first Get
4	GE_BOTTOM	Go to last Get
5	GE_ENTER	Get edit normal end
6	GE_WRITE	Terminate READ state with Get save
7	GE_ESCAPE	Terminate READ state without Get save
8	GE_WHEN	WHEN clause unsatisfied

Getexit.ch contains manifest constants for the Get:ExitState values.

```
While ( get:ExitState == GE_NOEXIT )
      GetApplyKey( oGet, Inkey(0) )

End
```

HasFocus

Contains a logical value that indicates if the Get object has input focus. Get:HasFocus contains true (.T.) if the Get object has input focus; otherwise it contains false (.F.).

```
If (oGet:HasFocus)
   .
   .
   .

Endif
```

Minus **(Assignable)**

Contains a logical value indicating that a minus sign (-) has been added to the editing buffer. Get:Minus is set to true (.T.) only when the Get object is numeric type, the current value of the editing buffer is zero and the last change to the editing buffer was the addition of the minus sign. It is cleared when any change is made to the buffer.

```
If (oGet:Minus)
   .
   .
   .

Endif
```

Name **(Assignable)**

Contains a character string representing the name of the Get variable. This value is optional and can be assigned at your discretion. With Get objects created using the standard @...GET command, Get:Name always contains the Get variable name.

The Get object itself ignores this variable. It is used by the standard READ command and READMODAL() function to implement the READVAR() function (see the READVAR() entry in this chapter).

```
oGet:Name := "GetObject1"
```

Original

Contains a value of any data type that is a copy of the value in the Get variable at the time the Get object acquired input focus. This value implements the Get:Undo message. Get:Original is meaningful only while the Get has input focus. At all other times, it contains NIL.

```
cTempVar := cOriginalVar := oGet:Original
```

Picture **(Assignable)**

Contains a character value defining the PICTURE string that controls formatting and editing for the Get object. See the @...GET entry in this chapter for more information on PICTURE strings.

```
oGet:Picture := "@R (999)999-9999"
```

Pos

Contains a numeric value indicating the position of the cursor within the editing buffer. Get:Pos is meaningful only when the Get object has input focus. At all other times, it contains NIL.

```
oGet:SetFocus()
If (oGet:Pos == 3)
   .
   .
   .

Endif
```

PostBlock **(Assignable)**

Contains an optional code block that validates a newly entered value. If present, the Get:PostBlock should contain an expression that evaluates to true (.T.) for a legal value and false (.F.) for an illegal value. For Get objects created with the standard @...GET...VALID command, Get:PostBlock is assigned the expression specified in the VALID clause.

The Get object itself ignores this variable. It is used by the standard READ command to implement the VALID clause.

During postvalidation the Get:PostBlock is passed a reference to the current Get object as an argument.

Note: This differs from the CA-Clipper 5.0 behavior where the Get:PostBlock was passed the updated value of the Get variable, and a logical value representing whether the Get object had been edited.

```
oGet:PostBlock := {|x| x != 0.00 }
```

PreBlock **(Assignable)**

Contains an optional code block that decides whether editing should be permitted. If present, the Get:PreBlock should evaluate to true (.T.) if the cursor enters the editing buffer; otherwise, it should evaluate to false (.F.). For Get objects created with the standard @...GET...WHEN command, Get:PreBlock is initialized with the expression specified in the WHEN clause.

The Get object itself ignores this variable. It is used by the standard READ command to implement the WHEN clause.

During prevalidation, the Get:PreBlock is passed a reference to the current Get object as an argument.

Note: This behavior differs from CA-Clipper 5.0 where no arguments were passed to Get:PreBlock when it was run during prevalidation.

```
oGet:PreBlock := {|x| x == 0.00 }
```

Reader

Contains a code block to implement special READ behaviors for any Get. If Get:Reader contains a code block, READMODAL() evaluates that block to READ the Get (the Get object is passed as an argument to the block). The block may in turn call any desired function to provide custom editing of the Get object. If Get:Reader does not contain a code block, READMODAL() uses a default read procedure (GetReader()) for the Get object.

Note that Get:Reader allows particular Get objects to have specialized READ behaviors without changing the standard READMODAL() function. This preserves compatibility for Gets which are to be handled in the customary way and also eliminates potential conflicts between different extensions to the GET/READ system.

```
oGet:Reader := { |oGet| MyRead( oGet ) }
FUNCTION MyRead( oGet )
  LOCAL cTemp, nKey := 0

  // Set Input Focus
  oGet:SetFocus()

  // Set cursor position to rightmost column
  SETPOS( oGet:row, oGet:Col + LEN( oGet:Buffer ) - 1 )

  // Store initial value of GET variable to oGet:Buffer
  oGet:Buffer := STR( oGet:VarGet() )

  // Store Get object buffer to a temporary variable
  cTemp := IF( oGet:VarGet() == 0, "", ALLTRIM( oGet:Buffer ) )

  // Key processing loop
  WHILE ( nKey := INKEY(0) ) != K_ENTER         ;
     .AND. nKey != K_UP                         ;
     .AND. nKey != K_DOWN                       ;
     .AND. nKey != K_ESC                        ;
     .AND.  LEN( cTemp ) < LEN( oGet:Buffer )   ;
     .AND. ( CHR(nKey) $ "0123456789-" .OR. nKey == K_DEL )

     IF CHR( nKey ) == "-"
        cTemp := IF(VAL(cTemp) != 0,;
                 ALLTRIM(STR(VAL( cTemp ) * -1 )), "")

     ELSEIF nKey == K_DEL
        cTemp := ""

     ELSE
        cTemp := cTemp + CHR( nKey )

     ENDIF

     IF LEN( cTemp ) > 0
        oGet:Buffer := PADL(cTemp,LEN(oGet:Buffer))

     ELSE
        oGet:Buffer := PADL("0", LEN(oGet:Buffer))

     ENDIF
        oGet:display()

     // Reset cursor position to rightmost column
     SETPOS(oGet:row, oGet:Col + LEN(oGet:Buffer) - 1)

  END
```

```
   // Store new value to GET variable
   IF nKey != K_ESC
      oGet:VarPut(VAL(oGet:Buffer))

   ENDIF

   // Update oGet:sxitState
   IF nKey == K_UP
      oGet:ExitState := GE_UP

   ELSEIF nKey == K_DOWN
      oGet:ExitState := GE_DOWN

   ELSEIF nKey == K_ESC
      oGet:ExitState := GE_ESCAPE

   ELSE
      oGet:ExitState := GE_ENTER

   ENDIF

   // Kill Input focus
   oGet:killFocus()

RETURN (NIL)
```

Rejected

Contains a logical value indicating whether the last character specified by a Get:insert or Get:overStrike message was placed in the editing buffer. Get:rejected contains true (.T.) if the last character was rejected; otherwise it contains false (.F.). Note that any subsequent text entry message resets this variable.

```
If (oGet:Rejected)
   .
   .
   .

Endif
```

Row **(Assignable)**

Contains a numeric value defining the screen row where the Get displays.

```
oGet:Row := 10
```

SubScript **(Assignable)**

Contains an array of numeric values representing the subscripts of a GET array element. Each element of Get:subscript represents a dimension of the GET array. For example:

```
@ 1,1 GET aTestA[4]        // Get:SubScript contains {4}
@ 1,1 GET aTestB[3,6]      // Get:SubScript contains {3,6}
```

If the GET does not involve an array, Get:SubScript contains NIL.

```
If ( oGet:SubScript <> NIL )
   for i := 1 to len(oGet:SubScript)
       name += "[" + ltrim( str( oGet:SubScript[i] ) ) + "]"

   next

Endif
```

Type

Contains a single letter representing the data type of the Get variable. For more information on the values that represent data types, refer to the VALTYPE() entry in this chapter.

```
If (oGet:Type == "N" .and. (cKey == "." .or. cKey == ","))
   oGet:ToDecPos()

Endif
```

TypeOut

Contains a logical value indicating whether the most recent message attempted to move the cursor out of the editing buffer, or if there are no editable positions in the buffer. Get:TypeOut contains true (.T.) if the last message satisfied this condition. Note, Get:TypeOut is reset by any message that moves the cursor.

```
If (oGet:TypeOut)
   oGet:ExitState := GE_ENTER

Endif
```

Assign() --> self

Assigns the value in the editing buffer to the Get variable by evaluating Get:Block with the buffer value supplied as its argument. This message is meaningful only when the Get object has input focus.

```
oGet:SetFocus()
If (oGet:Changed)
   oGet:Assign()

Endif
```

ColorDisp([<cColorString>]) --> self

```
   Get:ColorDisp() is a method that changes a Get object's colors and
   redisplays it.  It is exactly equivalent to assigning Get:ColorSpec
   and issuing Get:display().

   // White on cyan when highlighted, otherwise, black on cyan.
   oGet:ColorDisp("N/BG, +W/BG")
```

Display() --> self

Displays the Get on the screen. If the Get object has input focus, the Get:Buffer is displayed in its selected color and the cursor is placed at the screen location corresponding to the current editing position within the buffer. If the Get does not have input focus, the Get:Block is evaluated and the result displays in the Get unselected color with no cursor.

```
oGet:Display()
```

KillFocus() --> self

Takes input focus away from the Get object. Upon receiving this message, the Get object redisplays its editing buffer and discards its internal state information.

```
oGet:KillFocus()
```

CHR()
Convert an ASCII code to a character value

Syntax

CHR(<nCode>) --> cChar

Arguments

<nCode> is an ASCII code in the range of zero to 255.

Returns

CHR() returns a single character value whose ASCII code is specified by <nCode>.

Description

CHR() is a numeric conversion function that converts an ASCII code to a character. It is the inverse of ASC(). CHR() serves a number of common tasks including:

- Sending control codes and graphics characters to the screen or printer
- Ringing the bell
- Converting INKEY() return values to characters
- Stuffing the keyboard buffer

Notes

- **The null character:** CHR(0) has a length of one and is treated like any other character. This lets you send it to any device or file, including a database file.

Examples

- These examples illustrate CHR() with various arguments:

```
? CHR(72)                   // Result: H
? CHR(ASC("A") + 32)        // Result: a
? CHR(7)                    // Result: bell sounds
```

- These lines of code show the difference between a null string and the null character:

```
? LEN("")                   // Result: 0
? LEN(CHR(0))               // Result: 1
```

Files: Library is CLIPPER.LIB.

See Also: ASC() INKEY() Inkey codes

Reset() --> self

Resets the Get object's internal state information. This includes resetting the editing buffer to reflect the current value of the Get variable and setting the cursor position to the first editable position within the buffer. This message has meaning only when the Get object has input focus.

```
oGet:Reset()
```

SetFocus() --> self

Gives input focus to the Get object. Upon receiving this message, the Get object creates and initializes its internal state information, including the exported instance variables: Get:Buffer, Get:pos, Get:DecPos, and Get:original. The contents of the editing buffer are then displayed in the Get's selected color.

```
oGet:SetFocus()
```

Undo() --> self

Sets the Get variable back to the value it had when the Get acquired input focus. This message has meaning only while the Get has input focus.

The effect of the Get:Undo() message is equivalent to assigning the Get variable from the saved value in Get:original then sending the Get:reset() message.

oGet:Undo()

UnTransform() --> xValue

Converts the character value in the editing buffer back to the data type of the original variable. Get:Assign() is equivalent to Get:VarPut (Get:UnTransform()).

```
nNumber := oGet:UnTransform()
```

UpDateBuffer() --> self

Sets the editing buffer to reflect the current value of the Get variable, and redisplays the Get. This message has meaning only while the Get has input focus.

```
oGet:UpDateBuffer()
```

VarGet() --> GetVarValue

Returns the current value of the Get variable. For simple Get variables this is equivalent to executing Get:Block:

```
aGet:VarGet() == EVAL(aGet:Block)
```

However, if the Get variable is an array element, EVAL(aGet:Block) will not return the value of the Get variable; in this case, you must use VarGet(). An example of VarGet() may be found in the READMODAL() function, defined in Getsys.prg.

VarPut() --> Value

Sets the Get variable to the passed value. For simple Get variables this is equivalent to executing Get:Block with an argument:

```
aGet:VarPut(aValue) == EVAL(aGet:Block, aValue)
```

However, if the Get variable is an array element, EVAL(aGet:Block, aValue) will not set the value of the Get variable; in this case use of VarPut() is required.

End() --> self

Moves the cursor to the rightmost editable position within the editing buffer.

oGet:End()

Home() --> self

Moves the cursor to the leftmost editable position within the editing buffer.

oGet:Home()

Left() --> self

Moves the cursor left to the nearest editable position within the editing buffer. If there is no editable position to the left, the cursor position is left unchanged.

oGet:Left()

Right() --> self

Moves the cursor right to the nearest editable position within the editing buffer. If there is no editable position to the right, the cursor position is left unchanged.

oGet:Right()

ToDecPos() --> self

Moves the cursor to the immediate right of the decimal point position in the editing buffer. This message is only meaningful when editing numeric values.

```
oGet:ToDecPos()
```

WordLeft() --> self

Moves the cursor one word to the left within the editing buffer. If the cursor is already at the leftmost editable position, it is left unchanged.

oGet:WordLeft()

WordRight() --> self

Moves the cursor one word to the right within the editing buffer. If the cursor is already at the rightmost editable position, it is left unchanged.

oGet:WordRight()

BackSpace() --> self

Deletes the character to the left of the cursor moving the cursor one position to the left. If the cursor is already at the leftmost editable position in the editing buffer, this message has no effect.

oGet:BackSpace()

Delete() --> self

Deletes the character under the cursor.

```
oGet:Delete()
```

DelEnd() --> self

Deletes from the current character position to the end of the Get, inclusive.

```
oGet:DelEnd()
```

DelLeft() --> self

Deletes the character to the left of the cursor.

```
oGet:DelLeft()
```

DelRight() --> self

Deletes the character to the right of the cursor.

```
oGet:DelRight()
```

DelWordLeft() --> self

Deletes the word to the left of the cursor.

```
oGet:DelWordLeft()
```

DelWordRight() --> self

Deletes the word to the right of the cursor.

```
oGet:DelWordRight()
```

Insert(<cChar>) --> self

Inserts <cChar> into the editing buffer at the current cursor position, shifting the existing contents of the buffer to the right. The cursor is then placed one position to the right of the inserted string.

```
oGet:Insert("a")
```

OverStrike(<cChar>) --> self

Puts <cChar> into the editing buffer at the current cursor position, overwriting the existing contents of the buffer. The cursor is placed one position to the right of the inserted string.

```
oGet:Overstrike("a")
```

---------- Class Functions ---------- Exported Instance Variables ---------- Exported Methods Cursor Movement Methods Miscellaneous Methods

TBrowse
Provides objects for browsing table-oriented data

Description

A TBrowse object is a general purpose browsing mechanism for table-oriented data. TBrowse objects provide a sophisticated architecture for acquiring, formatting, and displaying data. Data retrieval and file positioning are performed via user-supplied code blocks, allowing a high degree of flexibility and interaction between the browsing mechanism and the underlying data source. The format of individual data items can be precisely controlled via the TBColumn data retrieval code blocks; overall display formatting and attributes can be controlled by sending appropriate messages to the TBrowse object.

A TBrowse object relies on one or more TBColumn objects. A TBColumn object contains the information necessary to define a single column of the browse table (see TBColumn class in this chapter).

During operation, a TBrowse object retrieves data by evaluating code blocks. The data is organized into rows and columns and displayed within a specified rectangular region of the screen. The TBrowse object maintains an internal browse cursor. The data item on which the browse cursor rests is displayed in a highlighted color. (The actual screen cursor is also positioned to the first character of this data item.)

Initially, the browse cursor is placed on the data item at the top left of the browse display. Messages can then be sent to the TBrowse object to navigate the displayed data, causing the browse cursor to move. These messages are normally sent in response to user keystrokes.

New data is automatically retrieved as required by navigation requests. When navigation proceeds past the edge of the visible rectangle, rows or columns beyond that edge are automatically brought into view. When new rows are brought into view, the underlying data source is repositioned by evaluating a code block.

Note: TBrowse objects do not clear the entire window before output during redisplay operations. Part of the window may still be cleared when data from the existing display is scrolled.

Examples

For fully operational examples of TBrowse objects, refer to the TBrowse introduction chapter in the Programming and Utilities guide and to the examples located in \CLIPPER5\SOURCE\TBROW subdirectories.

See Also: BROWSE()* DBEDIT()* SETCOLOR() TBColumn

TBrowseNew(<nTop>, <nLeft>, <nBottom>, <nRight>)
--> objTBrowse

Returns a new TBrowse object with the specified coordinate settings. The TBrowse object is created with no columns and no code blocks for data positioning. These must be provided before the TBrowse object can be used.

```
oTB := TBrowseNew( 3, 6, MaxRow() - 4, MaxCol() - 2)
```

TBrowseDB(<nTop>, <nLeft>, <nBottom>, <nRight>)
--> objTBrowse

Returns a new TBrowse object with the specified coordinate settings and default code blocks for data source positioning within database files. The default code blocks execute the GO TOP, GO BOTTOM, and SKIP operations.

Note that TBrowseDB() creates an object with no column objects. To make the TBrowse object usable, you must add a column for each field to be displayed.

```
oTB := TBrowseDB( 3, 6, MaxRow() - 4, MaxCol() - 2)
```

AutoLite **(Assignable)**

Contains a logical value. When autoLite is set to true (.T.), the stabilize method automatically highlights the current cell as part of stabilization. The default for autoLite is true (.T.).

```
oTB:AutoLite := .F.
```

Cargo **(Assignable)**

Contains a value of any data type provided as a user-definable slot. TBrowse:Cargo allows arbitrary information to be attached to a TBrowse object and retrieved later.

```
oTB:Cargo := Date()
```

ColCount

Contains a numeric value indicating the total number of data columns in the browse. For each column, there is an associated TBColumn object.

```
nCols := oTB:ColCount
```

ColorSpec **(Assignable)**

Contains a character string defining a color table for the TBrowse display. As a default, the current SETCOLOR() value is copied into this variable when the TBrowse object is created (see the SETCOLOR() reference in this chapter).

```
oTB:ColorSpec := "W/B*, N/BG, R+/B*, W/B*," +;
                 "GR+/B*, W+/B*, W+/GB, W+/B*"
```

ColPos **(Assignable)**

Contains a numeric value indicating the data column where the browse cursor is currently located. Columns are numbered from one, starting with the leftmost column.

```
nColPos := oTB:ColPos
```

ColSep **(Assignable)**

Contains a character value that defines a column separator for TBColumns that do not contain a column separator of their own (see TBColumn class in this chapter for more information).

```
oTB:ColSep := CHR(32) + CHR(179) + CHR(32)
```

FootSep **(Assignable)**

Contains a character value that defines a footing separator for TBColumns not containing a footing separator of their own (see TBColumn class in this chapter).

```
oTB:FootSep := CHR(205) + CHR(207) + CHR(205)
```

Freeze **(Assignable)**

Contains a numeric value that defines the number of data columns frozen on the left side of the display. Frozen columns are always visible, even when other columns are panned off the display.

```
oTB:Freeze := 1
```

GoBottomBlock **(Assignable)**

Contains a code block executed in response to the TBrowse:GoBottom() message. This block is responsible for repositioning the data source to the last record displayable in the browse. Typically the data source is a database file, and this block contains a call to a user-defined function that executes a GO BOTTOM command.

```
oTB:GoBottomBlock := {|| GoBottomDB(@nLine)}

FUNCTION GoBottomDB( nLine )
   DBGoBottom()
   nLine := LAST_LINE
RETURN (NIL)
```

GoTopBlock **(Assignable)**

Contains a code block that is executed in response to the TBrowse:GoTop() message. This block is responsible for repositioning the data source to the first record displayable in the browse. Typically the data source is a database file, and this block contains a call to a user-defined function that executes a GO TOP command.

```
oTB:GoTopBlock    := {|| GoTopDB(@nLine)}

FUNCTION GoTopDB( nLine )
   DBGoTop()
   nLine := 1
RETURN (NIL)
```

HeadSep **(Assignable)**

Contains a character value that defines a heading separator for TBColumns not containing a heading separator of their own (see TBColumn class in this chapter).

```
oTB:HeadSep := CHR(205) + CHR(209) + CHR(205)
```

HitBottom **(Assignable)**

Contains a logical value indicating whether an attempt was made to navigate beyond the end of the available data. TBrowse:HitBottom contains true (.T.) if an attempt was made; otherwise it contains false (.F.). During stabilization, the TBrowse object sets this variable if TBrowse:SkipBlock indicates it was unable to skip forward as many records as requested.

```
lLastRec := oTB:HitBottom
```

HitTop **(Assignable)**

Contains a logical value indicating whether an attempt was made to navigate past the beginning of the available data. TBrowse:HitTop contains true (.T.) if an attempt was made; otherwise it contains false (.F.). During stabilization, the TBrowse object sets this variable if TBrowse:SkipBlock indicates that it was unable to skip backward as many records as requested.

```
lFirstRec := oTB:HitTop
```

LeftVisible

Contains a numeric value indicating the position of the leftmost unfrozen column visible in the browse display. If every column is frozen in the display, TBrowse:LeftVisible contains zero.

```
nLeftCol := oTB:LeftVisible
```

nBottom **(Assignable)**

Contains a numeric value defining the bottom screen row used for the TBrowse display.

```
nBottomRow := oTB:nBottom
```

nLeft **(Assignable)**

Contains a numeric value defining the leftmost screen column used for the TBrowse display.

```
nLeftCol := oTB:nLeft
```

nRight **(Assignable)**

Contains a numeric value defining the rightmost screen column used for the TBrowse display.

```
nRightCol := oTB:nRight
```

nTop **(Assignable)**

Contains a numeric value defining the top screen row used for the TBrowse display.

```
nTopRow := oTB:nTop
```

RightVisible

Contains a numeric value indicating the position of the rightmost unfrozen column visible in the browse display. If all columns visible in the display are frozen, TBrowse:RightVisible contains zero.

```
nRightCol := oTB:RightVisible
```

RowCount

Contains a numeric value indicating the number of data rows visible in the browse display. Only data rows are included in the count. Rows occupied by headings, footings, or separators are not included.

```
nTotalDataRows := oTB:RowCount
```

RowPos **(Assignable)**

Contains a numeric value indicating the data row where the browse cursor is currently located. Data rows are numbered from one, starting with the topmost data row. Screen rows containing headings, footings or separators are not considered data rows.

```
nCurrentRow := oTB:RowPos
```

SkipBlock **(Assignable)**

Contains a code block that repositions the data source. During stabilization, this code block is executed with a numeric argument when the TBrowse object needs to reposition the data source. The numeric argument passed to the block represents the number of records to be skipped. A positive value means skip forward, and a negative value means skip backward. A value of zero indicates that the data source does not need to be repositioned, but the current record may need to be refreshed. Typically the data source is a database file, and this block calls a user-defined function that executes a SKIP command to reposition the record pointer.

The block must return the number of rows (positive, negative, or zero) actually skipped. If the value returned is not the same as the value requested, the TBrowse object assumes that the skip operation encountered the beginning or end of file.

```
 oTB:SkipBlock := {|nSkip| SkipDB(nSkip, @nLine)}

FUNCTION SkipDB( nRequest, nLine )
// nLine is a reference
LOCAL nActually := 0

IF nRequest == 0
   DBSKIP(0)

ELSEIF nRequest > 0 .AND. !EOF()
   WHILE nActually < nRequest
      IF nLine < LAST_LINE
         // This will print up to LAST_LINE of text
         // Some of them (or even all) might be empty
         ++nLine

      ELSE
         // Go to the next record
         DBSKIP(+1)
         nLine := 1

      ENDIF
      IF EOF()
         DBSKIP(-1)
         nLine := LAST_LINE
         EXIT

      ENDIF
      nActually++

   END

ELSEIF nRequest < 0
   WHILE nActually > nRequest
      // Go to previous line
      IF nLine > 1
         --nLine

      ELSE
         DBSKIP(-1)
         IF !BOF()
            nLine := LAST_LINE

         ELSE
            // You need this. Believe me!
            nLine := 1
            GOTO RECNO()
            EXIT
```

```
         ENDIF

      ENDIF
      nActually--

   END

ENDIF

RETURN (nActually)
```

Stable **(Assignable)**

Contains a logical value indicating whether the TBrowse object is stable. It contains true (.T.) if the TBrowse object is stable; otherwise, it contains false (.F.). The browse is considered stable when all data has been retrieved and displayed, the data source has been repositioned to the record corresponding to the browse cursor, and the current cell has been highlighted. When navigation messages are sent to the TBrowse object, TBrowse:Stable is set to false (.F.). After stabilization is performed using the TBrowse:Stabilize() message, TBrowse:Stable is set to true (.T.).

```
lIsStable := oTB:Stable
```

Down() --> self

Moves the browse cursor down one row. If the cursor is already on the bottom row, the display is scrolled up and a new row is brought into view. If the data source is already at the logical end of file and the browse cursor is already on the bottom row, TBrowse:HitBottom is set true (.T.).

```
oTB:Down()
```

end() --> self

Moves the browse cursor to the rightmost data column currently visible.

```
oTB:End()
```

GoBottom() --> self

Repositions the data source to logical bottom of file (by evaluating the TBrowse:GoBottomBlock), refills the display with the bottommost available data, and moves the browse cursor to the lowermost data row for which data is available. The pan position of the window is not changed.

oTB:GoBottom()

CMONTH()
Convert a date to a character month name

Syntax

CMONTH(<dDate>) --> cMonth

Arguments

<dDate> is the date value to convert.

Returns

CMONTH() returns the name of the month as a character string from a date value with the first letter uppercase and the rest of the string lowercase. For a null date value, CMONTH() returns a null string ("").

Description

CMONTH() is a date conversion function useful for creating formatted date strings that can be used in reports, labels, or screens.

Examples

. These examples illustrate CMONTH():

```
? CMONTH(DATE())              // Result: September
? CMONTH(DATE() + 45)         // Result: October
? SUBSTR(CMONTH(DATE()), 1, 3) +;
   STR(DAY(DATE()))           // Result: Sep 1
```

Files: Library is CLIPPER.LIB.

See Also: CDOW() DATE() DAY() MONTH() YEAR()

GoTop() --> self

Repositions the data source to the logical beginning of file (by evaluating the TBrowse:GoTopBlock), refills the display with the topmost available data, and moves the browse cursor to the uppermost data row for which data is available. The pan position of the window is not changed.

```
oTB:GoTop()
```

Home() --> self

Moves the browse cursor to the leftmost unfrozen column on the display.

```
oTB:Home()
```

Left() --> self

Moves the browse cursor left one data column. If the cursor is already on the leftmost displayed column, the display is panned and the previous data column (if there is one) is brought into view.

```
oTB:Left()
```

PageDown() --> self

Repositions the data source downward and refills the display. If the data source is already at the logical end of file (i.e., the bottommost available record is already shown), the browse cursor is simply moved to the lowermost row containing data. If the data source is already at the logical end of file and the browse cursor is already on the bottom row, TBrowse:HitBottom is set true (.T.).

```
oTB:PageDown()
```

PageUp() --> self

Repositions the data source upward and refills the display. If the data source is already at logical beginning of file (i.e., the topmost available record is already shown), the browse cursor is simply moved to the top data row. If the data source is already at logical beginning of file and the browse cursor is already on the first data row, TBrowse:HitTop is set true (.T.).

```
oTB:PageUp()
```

PanEnd() --> self

Moves the browse cursor to the rightmost data column, causing the display to be panned completely to the right.

```
oTB:PanEnd()
```

PanHome() --> self

Moves the browse cursor to the leftmost data column, causing the display to be panned all the way to the left.

```
oTB:PanHome()
```

PanLeft() --> self

Pans the display without changing the browse cursor, if possible. When the screen is panned with TBrowse:PanLeft(), at least one data column out of view to the left is brought into view, while one or more columns are panned off screen to the right.

```
oTB:PanLeft()
```

PanRight() --> self

Pans the display without changing the browse cursor, if possible. When the screen is panned with TBrowse:PanRight(), at least one data column out of view to the right is brought into view, while one or more columns are panned off the screen to the left.

```
oTB:PanRight()
```

Right() --> self

Moves the browse cursor right one data column. If the cursor is already at the right edge, the display is panned and the next data column (if there is one) is brought into view.

```
oTB:Right()
```

Up() --> self

Moves the browse cursor up one row. If the cursor is already on the top data row, the display is scrolled down and a new row is brought into view. If the data source is already at the logical beginning of file and the browse cursor is already on the top data row, TBrowse:HitTop is set true (.T.).

```
oTB:Up()
```

AddColumn(<objColumn>) --> self

Adds a new TBColumn object to the TBrowse object and TBrowse:ColCount is increased by one.

```
oTB:AddColumn(oTBCol)
```

ColorRect(<aRect>, <aColors>) --> self

Directly alters the color of a rectangular group of cells. <aRect> is an array of four numbers (top, left, bottom, and right). The numbers refer to cells within the data area of the browse display, not to screen coordinates. <aColors> is an array of two numbers. The numbers are used as indexes into the color table for the browse. These colors will become the normal and highlighted colors for the cells within the specified rectangle.

Cells that are colored using colorRect retain their color until they are scrolled off the screen up or down. Horizontal panning has no affect on these colors and, in fact, cells that are currently off screen left or right can be colored even if they are not visible.

This example colors the entire virtual window (on and off screen):

```
aRect := {1, 1, browse:RowCount, browse:ColCount}
browse:ColorRect( aRect, {2, 1} )
```

ColWidth(<nColumn>) --> nWidth

Returns the display width of column number <nColumn> as known to the browse. If <nColumn> is out of bounds or not supplied or not a number, the method returns zero.

```
nFirstColWidth := oTB:ColWidth(1)
```

Configure() --> self

Causes the TBrowse object to reexamine all instance variables and TBColumn objects, reconfiguring its internal settings as required. This message can force reconfiguration when a TBColumn object is modified directly.

```
oTB:Configure()
```

DeHilite() --> self

Causes the current cell (the cell to which the browse cursor is positioned) to be dehighlighted. This method is designed for use when TBrowse:AutoLite is set to false (.F.).

```
oTB:DeHilite()
```

DelColumn(<nPos>) --> objColumn

This new method allows a column to be deleted from a browse. The return value is a reference to the column object being deleted so that the column object may be preserved.

```
oFirstTBCol := oTB:DelColumn(1)
```

ForceStable()

Performs a full stabilization of the TBrowse. It is analogous to the following code, only slightly faster:

```
DO WHILE .NOT. oBrowse:Stabilize()
ENDDO

oTB:ForceStable()
```

GetColumn(<nColumn>) --> objColumn

Returns the TBColumn object specified by <nColumn>.

```
oFirstTBCol := oTB:GetColumn(1)
```

Hilite() --> self

Causes the current cell (the cell to which the browse cursor is positioned) to be highlighted. This method is designed for use when TBrowse:AutoLite is set to false (.F.).

```
oTB:Hilite()
```

InsColumn(<nPos>, <objColumn>) --> objColumn

This method allows a column object to be inserted into the middle of a browse. The return value is a reference to the column object being inserted.

```
oInsertTBCol := oTB:InsColumn(2, oInsertTBCol)
```

Invalidate() --> self

TBrowse:Invalidate() causes the next stabilization of the TBrowse object to redraw the entire TBrowse display, including headings, footings, and all data rows. Note that sending this message has no effect on the values in the data rows; it simply forces the display to be updated during the next stabilization. To force the data to be refreshed from the underlying data source, send the TBrowse:RefreshAll() message.

```
oTB:Invalidate()
```

RefreshAll() --> self

Internally marks all data rows as invalid, causing them to be refilled and redisplayed during the next stabilize loop.

```
oTB:RefreshAll()
```

RefreshCurrent() --> self

Internally marks the current data row as invalid, causing it to be refilled and redisplayed during the next stabilize loop.

```
oTB:RefreshCurrent()
```

SetColumn(<nColumn>, <oColumnNew>) --> oColumnCurrent

Replaces the TBColumn object indicated by <nColumn> with the TBColumn object specified by <oColumnNew>. The value returned is the current TBColumn object.

```
oCurrentTBCol := oTB:SetColumn(2, oNewTBCol)
```

Stabilize() --> lStable

Performs incremental stabilization. Each time this message is sent, some part of the stabilization process is performed. Stabilization is performed in increments so that it can be interrupted by a keystroke or other asynchronous event.

If the TBrowse object is already stable, a value of true (.T.) is returned. Otherwise, a value of false (.F.) is returned indicating that further stabilize messages should be sent. The browse is considered stable when all data has been retrieved and displayed, the data source has been repositioned to the record corresponding to the browse cursor, and the current cell has been highlighted.

```
lIsStable := oTB:stabilize()
```

---------- Class Function ---------- Exported Instance Variables

TBColumn
Provides column objects for TBrowse objects

Description

A TBColumn object is a simple object containing the information needed to fully define one data column of a TBrowse object (see the TBrowse reference in this chapter). TBColumn objects have no methods, only exported instance variables.

Examples

. This example is a code fragment that creates a TBrowse object and adds some TBColumn objects to it:

```
USE Customer NEW
//

// Create a new TBrowse object
objBrowse := TBrowseDB(1, 1, 23, 79)
//

// Create some new TBColumn objects and
// add them to the TBrowse object
objBrowse:addColumn(TBColumnNew( "Customer", ;
      {|| Customer->Name} ))
objBrowse:addColumn(TBColumnNew( "Address", ;
      {|| Customer->Address} ))
objBrowse:addColumn(TBColumnNew( "City", ;
      {|| Customer->City} ))
.
. <statements to actually browse the data>
.
CLOSE Customer
```

For more information on TBrowse, refer to the Introduction to TBrowse chapter in the Programming and Utilitites guide.

For fully operational examples of TBrowse objects, see also the TBROW subdirectory located in \CLIPPER5\SOURCE\TBROW.

See Also: BROWSE()* DBEDIT()* TBrowse

TBColumnNew(<cHeading>, <bBlock>) --> objTBColumn

Returns a new TBColumn object with the specified heading and data retrieval block. Other elements of the TBColumn object can be assigned directly using the syntax for assigning exported instance variables.

```
oTBCol := TBColumnNew( "City", {|| mydbf->city })
```

Block **(Assignable)**

Contains a code block that retrieves data for the column. Any code block is valid, and no block arguments are supplied when the block is evaluated. The code block must return the appropriate data value for the column.

```
oTBCol:Block := {|| mydbf->city }
```

Cargo **(Assignable)**

Contains a value of any data type provided as a user-definable slot, allowing arbitrary information to be attached to a TBColumn and retrieved later.

```
oTBCol:Cargo := {|x| IF(x == NIL, mydbf->city,;
                                  mydbf->city := x)}
```

ColorBlock **(Assignable)**

Contains an optional code block that determines the color of data items as they are displayed. If present, this block is executed each time a new value is retrieved via the TBColumn:Block (the data retrieval block). The newly retrieved data value is passed as an argument to the TBColumn:ColorBlock, which must return an array containing two numeric values. The values returned are used as indexes into the color table of the TBrowse object as described in the TBColumn:defColor reference below.

The TBColumn:ColorBlock allows display colors for data items based on the value of the data being displayed. For example, negative numbers may be displayed in a different color than positive numbers.

```
oTBCol:ColorBlock := {|x| IF(x < 0, {5,2},;
                         IF(x > 0, {8,2}, {4,2}))}
```

ColSep **(Assignable)**

Contains an optional character string that draws a vertical separator to the left of this column if there is another column to the left of it. If no value is supplied for TBColumn:ColSep, the value contained in TBrowse:ColSep is used instead.

```
oTBCol:ColSep := CHR(32) + CHR(179) + CHR(32)
```

DefColor **(Assignable)**

Contains an array of two numeric values used as indexes into the color table in the TBrowse object. The first value determines the unselected color which displays headings, footings, etc. It also displays data values when the browse cursor is not on the data value being displayed. The second value determines the selected color. The selected color displays the current browse cell.

The default value for TBColumn:DefColor is {1, 2}. This causes the first two colors in the TBrowse color table to be used for unselected and selected, respectively. Note that colors set using TBColumn:ColorBlock override those set by TBColumn:DefColor.

```
oTBCol:DefColor := {5,7}
```

Footing **(Assignable)**

Contains a character value that defines the footing for this data column. The ";" character will insert a line break.

```
oTBCol:Footing := "Computer Associates Int'l"
```

FootSep **(Assignable)**

Contains a character value that draws a horizontal line between the data values and the footing. If it does not contain a character value: TBrowse:FootSep is used instead.

```
oTBCol:FootSep := CHR(205) + CHR(207) + CHR(205)
```

Heading **(Assignable)**

Contains a character value that defines the heading for this data column. The ";" character will insert a line break.

```
oTBCol:Heading := "Customer Name"
```

HeadSep **(Assignable)**

Contains an optional character string that draws a horizontal separator between the heading and the data values. If it does not contain a character value, the TBrowse:HeadSep is used instead.

```
oTBCol:HeadSep := CHR(205) + CHR(209) + CHR(205)
```

Picture **(Assignable)**

Contains an optional picture string that formats the data column.

```
oTBCol:Picture := "@R (999)-999-9999"
```

Width **(Assignable)**

Contains a numeric value that defines the display width for the column. If TBColumn:Width is not explicitly set, the width of the column will be the greater of: the length of the heading, the length of the footing, or the length of the data at the first evaluation of TBColumn:Block.

If this instance variable is explicitly set, the width of the column will be TBColumn:Width. Displayed headings, footings, and data will all be truncated to this width when necessary. The width of the displayed data will be the length at the first evaluation of TBColumn:Block for all data types other than character. Character data will be extended to TBColumn:Width for display purposes.

```
oTBCol:Width := 10
```

COL()
Return the screen cursor column position

Syntax

COL() --> nCol

Returns

COL() returns an integer numeric value. The range of the return value is zero to MAXCOL().

Description

COL() is a screen function that returns the current column position of the cursor. The value of COL() changes whenever the cursor position changes on the screen. Both console and full-screen commands can change the cursor position. In addition, COL() is automatically set to zero whenever a CLEAR, CLEAR SCREEN, or CLS command is executed.

Use COL() to position the cursor to a column relative to the current column. It is generally used in combination with ROW() and all variations of the @ command. In particular, use COL() and ROW() to create screen position-independent procedures or functions that pass the upper left row and column as parameters.

If DEVICE is SET TO PRINTER, all the output of @...SAY commands is directed to the printer and PROW() and PCOL() are updated instead of ROW() and COL(). Use these functions when you need to determine the position of the printhead.

Examples

. This example displays a Customer name beginning at column 10. The customer's account status is then displayed to the right of the last character of the customer name using COL():

```
USE Sales NEW

CLS
@ 1, 10 SAY "Customer Name: " + TRIM(Customer)
@ ROW(), COL() + 1 SAY "Account status: " + Status
```

Files: Library is CLIPPER.LIB.

See Also: @...SAY MAXCOL() PCOL() PROW() QOUT() ROW()

COLORSELECT()
Activate attribute in current color settings

Syntax

COLORSELECT(<nColorIndex>) --> NIL

Returns

Always returns NIL.

Arguments

<nColorIndex> is a number corresponding to the oridnal positions in the current list of color attributes, as set by SETCOLOR().

Description

COLORSELECT() activates the specified color pair from the current list of color attributes (established by SETCOLOR()). Manifest constants for <nColorIndex> are defined in COLOR.CH:

COLOR.CH constants

Constant	Value
CLR_STANDARD	0
CLR_ENHANCED	1
CLR_BORDER	2
CLR_BACKGROUND	3
CLR_UNSELECTED	4

COLORSELECT() does not alter the current SET Color setting.

This table describes the scope of the CA-Clipper color settings affected by SETCOLOR():

Color settings

Setting	Scope
Standard	All screen output commands and functions
Enhanced	Gets and selection highlights
Border	Screen border (not supported on EGA and VGA monitors)
Background	Not supported
Unselected	Unselected Gets

Examples

. This example demonstrates use of COLORSELECT() with the COLOR.CH manifest constants:

```
USE Sales NEW
? SETCOLOR()                  // displays "W/B,N/B,W/N,W/N,W/N"
                              // in white on blue

COLORSELECT(CLR_ENHANCED)     // enhanced is active color pair
? "I'm black and blue"        // displayed in black on blue
COLORSELECT(CLR_STANDARD)     // restore standard color
```

Files: Library is CLIPPER.LIB, header file is Color.ch.

See Also: SETCOLOR()

CTOD()
Convert a date string to a date value

Syntax

CTOD(<cDate>) --> dDate

Arguments

<cDate> is a character string consisting of numbers representing the month, day, and year separated by any character other than a number. The month, day, and year digits must be specified in accordance with the SET DATE format. If the century digits are not specified, the century is determined by the rules of SET EPOCH.

Returns

CTOD() returns a date value. If <cDate> is not a valid date, CTOD() returns an empty date.

Description

CTOD() is a character conversion function that converts a character string to a date. To initialize an empty date for date entry, specify <cDate> as a null string (""), SPACE(8), or " / / ".

CTOD() is used whenever you need a literal date value. Some examples are:

- Initializing a variable to a date value
- Specifying a literal date string as an argument of a RANGE clause of @...GET
- Specifying a literal date string in order to perform date arithmetic
- Comparing the result of a date expression to a literal date string
- REPLACEing a date field with a literal date string

CTOD() is the inverse of DTOC() which converts a date value to a character string in the format specified by SET DATE and SET CENTURY. DTOS() also converts a date value to a character string in the form yyyymmdd.

Examples

- This example uses CTOD() to initialize two date variables, using one as a GET and the other for RANGE validation:

```
SET CENTURY ON
dBegin := CTOD("01-26-1876")
dCurrent := CTOD("")
@ 10, 10 SAY "Enter date:" GET dCurrent ;
      RANGE dBegin, DATE()
READ
```

- This example uses CTOD() to create a date value within a FOR condition:

```
USE Inventory NEW
REPLACE ALL Inventory->Price WITH ;
   Inventory->Price * 1.1 FOR ;
   Inventory->InvDate < CTOD("10/10/90")
```

Files: Library is CLIPPER.LIB.

See Also: DATE() DTOC() DTOS() SET CENTURY SET DATE SET EPOCH

CURDIR()
Return the current DOS directory

Syntax

CURDIR([<cDrivespec>]) --> cDirectory

Arguments

<cDrivespec> specifies the letter of the disk drive to query. If not specified, the default is the current DOS drive.

Returns

CURDIR() returns the current DOS directory of the drive specified by <cDrivespec> as a character string without either leading or trailing backslash (\) characters.

If an error occurs, or the current directory of the specified drive is the root directory, CURDIR() returns a null string ("").

Description

CURDIR() is an environment function that gives you the name of the current DOS directory, ignoring the SET DEFAULT and SET PATH settings.

Examples

. These examples illustrate various CURDIR() results:

```
? CURDIR("E:")     // Result: null string--root directory
? CURDIR("C")      // Result: CLIPPER5\SOURCE
? CURDIR("C:")     // Result: CLIPPER5\SOURCE
? CURDIR()         // Result: null string--root directory
? CURDIR("A")      // Result: null string--drive not ready
```

. This example changes the current DOS directory to a new value if it does not match a specified directory:

```
IF CURDIR("C:") != "CLIPPER5\SOURCE"
   RUN CD \CLIPPER5\SOURCE
ENDIF
```

Files Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM.

See Also: FILE()

DATE()
Return the system date as a date value

Syntax

DATE() --> dSystem

Returns

DATE() returns the system date as a date value.

Description

DATE() is a date function that provides a means of initializing memory variables to the current date, comparing other date values to the current date, and performing date arithmetic relative to the current date.

The display format for dates is controlled by the SET DATE command. The default format is mm/dd/yy.

Examples

. These examples show the DATE() function used in various ways:

```
? DATE()                 // Result: 09/01/90
? DATE() + 30            // Result: 10/01/90
? DATE() - 30            // Result: 08/02/90
dDate := DATE()
? CMONTH(dDate)          // Result: September
```

Files: Library is CLIPPER.LIB.

See Also: CTOD() DTOC() DTOS() SET CENTURY SET DATE

DAY()
Return the day of the month as a numeric value

Syntax

DAY(<dDate>) --> nDay

Arguments

<dDate> is a date value to convert.

Returns

DAY() returns a number in the range of zero to 31 as an integer numeric value. If the month is February, leap years are considered. If the date argument is February 29 and the year is not a leap year, DAY() returns zero. If the date argument is empty, DAY() returns zero.

Description

DAY() is a date conversion function used to convert a date value to the day of a month. This function is used in combination with CMONTH() and YEAR() to format dates. In addition, it is often used in various date calculations.

Examples

. These examples show the DAY() function used several ways:

```
? DATE()                        // Result: 09/01/90
? DAY(DATE())                   // Result: 1
? DAY(DATE()) + 1               // Result: 2
? DAY(CTOD(""))                 // Result: 0
```

. This example uses DAY() in combination with CMONTH() and YEAR() to format a date value:

```
? CMONTH(DATE()) + STR(DAY(DATE())) +;
   "," + STR(YEAR(DATE()))      // Result: June 15, 1990
```

Files: Library is CLIPPER.LIB.

See Also: CDOW() CMONTH() DOW() MONTH() STR() YEAR()

DBAPPEND()
Add a new record

Syntax

DBAPPEND() --> NIL

Returns

DBAPPEND() always returns NIL.

Description

DBAPPEND() adds a new record to the database (.dbf) file associated with the current work area. If successfully added, each field in the record is set to the empty value for its data type and the new record becomes the current record.

DBAPPEND() performs the same function as the standard APPEND BLANK command. For more information, refer to the APPEND BLANK command.

Notes

- **Logical records:** DBAPPEND() does not respect logical visibility. That is, if the record is successfully added, it becomes the current record regardless of any index or filter condition.

- **Network environment:** For a shared database on a network, DBAPPEND() automatically places a record lock on the new record. If the record cannot be locked, it is not added and a recoverable runtime error occurs. The default CA-Clipper error handler handles this error by setting the system NETERR() status to true (.T.) and continuing execution. For more information, refer to the Network Programming chapter in the Programming and Utilities guide.

Examples

- The following example appends a blank record, checks for a network error and updates the data:

```
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
.
. <statements>
.
DBAPPEND()

IF !NETERR()
   Sales->FirstName := cFirst
   Sales->LastName := cLast
ELSE
   ? "Append operation failed"
   BREAK
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: APPEND BLANK

DBCLEARFILTER()
Clear a filter condition

Syntax

DBCLEARFILTER() --> NIL

Returns

DBCLEARFILTER() always returns NIL.

Description

DBCLEARFILTER() clears the logical filter condition, if any, for the current work area.

DBCLEARFILTER() performs the same function as the standard SET FILTER command with no expression specified. For more information, refer to the SET FILTER command.

Notes

- **Logical records:** DBCLEARFILTER() affects the logical visibility of records in the current work area. For more information, refer to DBSETFILTER() and the SET FILTER command.

Examples

- The following example sets a filter, LISTs data as filtered and then clears the filter:

```
USE Employee NEW
DBSETFILTER( {|| Age < 40}, "Age < 40" )
LIST Employee->Name
DBCLEARFILTER()
DBCLEARFILTER()
```

Files: Library is CLIPPER.LIB.

See Also: DBFILTER() DBSETFILTER() SET DELETED SET FILTER

DBCLEARINDEX()
Close all indexes for the current work area

Syntax

DBCLEARINDEX() --> NIL

Returns

DBCLEARINDEX() always returns NIL.

Description

DBCLEARINDEX() closes any active indexes for the current work area. Any pending index updates are written and the index files are closed.

DBCLEARINDEX() performs the same function as the standard SET INDEX command with no indexes specified. For more information, refer to the SET INDEX command.

Examples

. The following example clears index files if any are set:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "FIRSTNAM" )
DBSETINDEX( "LASTNAME" )
//
IF INDEXORD() > 0             // is there an index?
   DBCLEARINDEX()             // clear index files
ELSE
   COPY TO FILE TEMP SDF      // copy to SDF in natural
ENDIF                         // order
```

Files: Library is CLIPPER.LIB.

See Also: DBREINDEX() DBSETINDEX() DBSETORDER() SET INDEX

DBCLEARRELATION()
Clear active relations

Syntax

DBCLEARRELATION() --> NIL

Returns

DBCLEARRELATION() always returns NIL.

Description

DBCLEARRELATION() clears any active relations for the current work area.

DBCLEARRELATION() performs the same function as the standard SET RELATION TO command with no clauses specified. For more information, refer to the SET RELATION command.

Examples

. The following example sets a relation, lists data and then clears the relation:

```
USE Employee NEW
USE Department NEW INDEX Dept
//
SELECT Employee
DBSETRELATION("Department", ;
   {|| Employee->Dept}, "Employee->Dept")
LIST Employee->Name, Department->Name
DBCLEARRELATION()
```

Files: Library is CLIPPER.LIB.

See Also: DBSETRELAT() SET RELATION

DBCLOSEALL()
Close all occupied work areas

Syntax

DBCLOSEALL() --> NIL

Returns

DBCLOSEALL() always returns NIL.

Description

DBCLOSEALL() releases all occupied work areas from use. It is equivalent to calling DBCLOSEAREA() on every occupied work area. DBCLOSEAREA() has the same effect as the standard CLOSE DATABASES command. For more information, refer to the USE and CLOSE commands.

Examples

. The following example closes all work areas:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "SALEFNAM" )
DBSETINDEX( "SALELNAM" )
//
DBUSEAREA( .T., "DBFNTX", "Colls", "Colls", .T. )
DBSETINDEX( "COLLFNAM" )
DBSETINDEX( "COLLLNAM" )
//
DBSELECTAREA( "Sales" )      // select "Sales" work area
//
IF ( Sales->(DBSEEK(cLast)) )
   IF Sales->( DELETED() )
      IF RLOCK()
         Sales->( DBRECALL() )
         ? "Record deleted: ", Sales( DELETED() )
      ENDIF
   ENDIF
ELSE
   ? "Not found"
ENDIF
DBCLOSEALL()                 // close all work areas
```

Files: Library is CLIPPER.LIB.

See Also: CLOSE DBCLOSEAREA() DBUSEAREA() SELECT USE

DBCLOSEAREA()
Close a work area

Syntax

DBCLOSEAREA() --> NIL

Returns

DBCLOSEAREA() always returns NIL.

Description

DBCLOSEAREA() releases the current work area from use. Pending updates are written, pending locks are released, and any resources associated with the work area are closed or released. DBCLOSEAREA() is equivalent to the standard CLOSE command or the USE command with no clauses. For more information, refer to the USE and CLOSE commands.

Examples

. The following example closes a work area via an alias reference:

```
cLast := "Winston"

//
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "SALEFNAM" )
DBSETINDEX( "SALELNAM" )
//
DBUSEAREA( .T., "DBFNTX", "Colls", "Colls", .T. )
DBSETINDEX( "COLLFNAM" )
DBSETINDEX( "COLLLNAM" )
//
DBSELECTAREA( "Sales" )      // select "Sales" work area
//
IF ( Sales->(DBSEEK(cLast)) )
   IF Sales->( DELETED() ) .AND. Sales->( RLOCK() )
      Sales->( DBRECALL() )
      ? "Record deleted: ", Sales( DELETED() )
   ENDIF
ELSE
   ? "Not found"
   Colls->( DBCLOSEAREA() )
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: CLOSE DBCLOSEALL() DBCOMMIT() DBUSEAREA() USE

DBCOMMIT()
Flush pending updates

Syntax

DBCOMMIT() --> NIL

Returns

DBCOMMIT() always returns NIL.

Description

DBCOMMIT() causes all updates to the current work area to be written to disk. All updated database and index buffers are written to DOS and a DOS COMMIT request is issued for the database (.dbf) file and any index files associated with the work area.

DBCOMMIT() performs the same function as the standard COMMIT command except that it operates only on the current work area. For more information, refer to the COMMIT command.

Notes

- **Network environment:** DBCOMMIT() makes database updates visible to other processes. To insure data integrity, issue DBCOMMIT() before an UNLOCK operation. For more information, refer to the Network Programming chapter in the Programming and Utilities guide.

- DBCOMMIT() uses DOS interrupt 21h function 68h to perform the solid-disk write. It is up to the network operating system to properly implement this request. Check with the network vendor to see if this is supported.

Examples

- In this example, COMMIT is used to force a write to disk after a series of memory variables are assigned to field variables:

```
USE Sales EXCLUSIVE NEW
MEMVAR->Name := Sales->Name
MEMVAR->Amount := Sales->Amount
//
@ 10, 10 GET MEMVAR->Name
@ 11, 10 GET MEMVAR->Amount
READ
//
IF UPDATED()
   APPEND BLANK
   REPLACE Sales->Name WITH MEMVAR->Name
   REPLACE Sales->Amount WITH MEMVAR->Amount
   Sales->( DBCOMMIT() )
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: CLOSE COMMIT DBCLOSEALL() DBCOMMITALL() DBUNLOCK()

DBCOMMITALL()
Flush pending updates in all work areas

Syntax

DBCOMMITALL() --> NIL

Returns

DBCOMMITALL() always returns NIL.

Description

DBCOMMITALL() causes all pending updates to all work areas to be written to disk. It is equivalent to calling DBCOMMIT() for every occupied work area.

For more information, refer to DBCOMMIT() and the COMMIT command.

Notes

. DBCOMMITALL() uses DOS interrupt 21h function 68h to perform the solid-disk write. It is up to the network operating system to properly implement this request. Check with the network vendor to see if this is supported.

Examples

. The following example writes all pending updates to disk:

```
cLast := "Winston"
//
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "SALEFNAM" )
DBSETINDEX( "SALELNAM" )
//
DBUSEAREA( .T., "DBFNTX", "Colls", "Colls", .T. )
DBSETINDEX( "COLLFNAM" )
DBSETINDEX( "COLLLNAM" )

DBSELECTAREA( "Sales" )      // select "Sales" work area

IF ( Sales->(DBSEEK(cLast)) )
   IF Sales->( DELETED() ) .AND. Sales( RLOCK() )
      Sales->( DBRECALL() )
      ? "Deleted record has been recalled."
   ENDIF
ELSE
   ? "Not found"
ENDIF
//
// processing done, write updates to disk and close
DBCOMMITALL()
DBCLOSEALL()
QUIT
```

Files: Library is CLIPPER.LIB.

See Also: CLOSE COMMIT DBCLOSEALL() DBCOMMIT() DBUNLOCK()

DBCREATE()
Create a database file from a database structure array

Syntax

DBCREATE(<cDatabase>, <aStruct>
[<cDriver>]) --> NIL

Arguments

<cDatabase> is the name of the new database file, with an optional drive and directory, specified as a character string. If specified without an extension (.dbf) is assumed.

<aStruct> is an array that contains the structure of <cDatabase> as a series of subarrays, one per field. Each subarray contains the definition of each field's attributes and has the following structure:

Field Definition Subarray

Position	Metasymbol	Dbstruct.ch
1	cName	DBS_NAME
2	cType	DBS_TYPE
3	nLength	DBS_LEN
4	nDecimals	DBS_DEC

<cDriver> specifies the replaceable database driver (RDD) to use to process the current work area. <cDriver> is name of the RDD specified as a character expression. If you specify <cDriver> as a literal value, you must enclose it in quotes.

Returns

DBCREATE() always returns NIL.

Description

DBCREATE() is a database function that creates a database file from an array containing the structure of the file. You may create the array programmatically or by using DBSTRUCT(). DBCREATE() is similar to the CREATE FROM command which creates a new database file structure from a structure extended file. Use CREATE or COPY STRUCTURE EXTENDED commands to create a structure extended file.

Before using DBCREATE(), you must first create the <aStruct> array and fill it with the field definition arrays according to the structure in Field Definition Subarray table (above). There are some specific rules for creating a field definition array, including:

- Specify all field attributes with a value of the proper data type for the attribute. The decimals attribute must be specified--even for non-numeric fields. If the field does not have a decimals attribute, specify zero.

- Specify the type attribute using the first letter of the data type as a minimum. Use longer and more descriptive terms for readability. For example, both "C" and "Character" can be specified as the type attribute for character fields.

- In CA-Clipper, character fields contain up to 64,000 characters. Unlike the CREATE FROM command, DBCREATE() does not use the decimals attribute to specify the high-order part of the field length. Specify the field length directly, regardless of its magnitude.

To make references to the various elements of the field definition subarray more readable, the header file called Dbstruct.ch is supplied which contains the #defines to assign a name to the array position for each field attribute. It is located in \CLIPPER5\INCLUDE.

Examples

. This example creates an empty array and then adds field definition subarrays using the AADD() function before creating People.dbf. You might use this technique to add field definitions to your structure array dynamically:

```
aDbf := {}
AADD(aDbf, { "Name", "C", 25, 0 })
AADD(aDbf, { "Address", "C", 1024, 0 })
AADD(aDbf, { "Phone", "N", 13, 0 })
//
DBCREATE("People", aDbf)
```

. This example performs the same types of actions but declares the structure array as a two-dimensional array, and then uses subscript addressing to specify the field definitions. It will be created using the DBFMDX RDD:

```
#include "Dbstruct.ch"
//
LOCAL aDbf[1][4]
aDbf[1][ DBS_NAME ] := "Name"
aDbf[1][ DBS_TYPE ] := "Character"
aDbf[1][ DBS_LEN ]  := 25
aDbf[1][ DBS_DEC ]  := 0
//
DBCREATE("Name", aDbf, "DBFMDX")
```

Files: Library is CLIPPER.LIB, header file is Dbstruct.ch.

See Also: AFIELDS()* COPY STRU EXTE CREATE FROM DBSTRUCT()

DBCREATEINDEX()
Create an index file

Syntax

DBCREATEINDEX(<cIndexName>, <cKeyExpr>,
<bKeyExpr>, [<lUnique>]) --> NIL

Arguments

<cIndexName> is a character value that specifies the filename of the index file to be created.

<cKeyExpr> is a character value that expresses the index key expression in textual form.

<bKeyExpr> is a code block that expresses the index key expression in executable form.

<lUnique> is an optional logical value that specifies whether a unique index is to be created. If <lUnique> is omitted, the current global _SET_UNIQUE setting is used.

Returns

DBCREATEINDEX() always returns NIL.

Description

DBCREATEINDEX() creates an index for the database (.dbf) file associated with the current work area. If the work area has active indexes, they are closed. After the new index is created, it becomes the controlling index for the work area and is positioned to the first logical record.

DBCREATEINDEX() performs the same function as the standard INDEX command. For more information, refer to the INDEX command.

Notes

- **Side effects:** DBCREATEINDEX() is guaranteed to create an index that, when made active, will impose the specified logical order on the database. The key expression is not necessarily evaluated at any particular time, by any particular means, or on any particular record or series of records. If the key expression relies on information external to the database file or work area, the effect is unpredictable. If the key expression changes the state of the work area (e.g., by moving to a different record or changing the contents of a record), the effect is unpredictable.

- **Evaluation context:** When the key expression is evaluated, the associated work area is automatically selected as the current work area before the evaluation; the previously selected work area is automatically restored afterward.

- **Network environment:** DBCREATEINDEX() creates the new index for the exclusive use of the current process.

Examples

- This example creates an index file, Name, indexed on the Name field:

```
USE Employees NEW
DBCREATEINDEX( "Name", "Name", { || Name })
```

Files: Library is CLIPPER.LIB.

See Also: DBREINDEX() DBSETINDEX() DBSETORDER() INDEX

DBDELETE()
Mark a record for deletion

Syntax

DBDELETE() --> NIL

Returns

DBDELETE() always returns NIL.

Description

DBDELETE() marks the current record as deleted. Records marked for deletion can be filtered using SET DELETED or removed from the file using the PACK command.

DBDELETE() performs the same function as the standard DELETE command with a scope of the current record. For more information, refer to the DELETE command.

Notes

- **Logical records:** If the global _SET_DELETED status is true (.T.), deleted records are not logically visible. That is, database operations which operate on logical records will not consider records marked for deletion. Note, however, that if _SET_DELETED is true (.T.) when the current record is marked for deletion, the record remains visible until it is no longer the current record.

- **Network environment:** For a shared database on a network, DBDELETE() requires the current record to be locked. For more information, refer to the Network Programming chapter of the Programming and Utilities guide.

Examples

- The following example deletes a record after a successful record lock:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "LASTNAME" )
//
IF ( Sales->(DBSEEK(cLast)) )
   IF Sales->( RLOCK() )
      Sales->( DBDELETE() )
      ? "Record deleted: ", Sales( DELETED() )
   ELSE
      ? "Unable to lock record..."
   ENDIF
ELSE
   ? "Not found"
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: DBRECALL() DELETE RECALL

DBEDIT()*
Browse records in a table layout

Syntax

```
DBEDIT([<nTop>], [<nLeft>],
   [<nBottom>], <nRight>],
   [<acColumns>],
   [<cUserFunction>],
   [<acColumnSayPictures> | <cColumnSayPicture>],
   [<acColumnHeaders> | <cColumnHeader>],
   [<acHeadingSeparators> | <cHeadingSeparator>],
   [<acColumnSeparators> | <cColumnSeparator>],
   [<acFootingSeparators> | <cFootingSeparator>],
   [<acColumnFootings> | <cColumnFooting>]) --> NIL
```

Arguments

<nTop>, **<nLeft>**, **<nBottom>**, and **<nRight>** define the upper left and lower right coordinates of the DBEDIT() window. Row values can range from zero to MAXROW() and column positions can range from zero to MAXCOL(). If not specified, the default coordinates are 0, 0, MAXROW(), and MAXCOL().

<acColumns> is an array of character expressions containing database field names or expressions to use as column values for each row displayed. If this argument is not specified, DBEDIT() displays all fields in the current work area as columns.

<cUserFunction> is the name of a user-defined function that executes when an unrecognizable key is pressed or there are no keys pending in the keyboard buffer. Specify the function name as a character expression without parentheses or arguments. Note that the behavior of DBEDIT() is affected by the presence of this argument. Refer to the discussion below for more information.

<acColumnSayPictures> is a parallel array of picture clauses to format each column. Specifying <cColumnSayPicture> instead of an array displays all columns with the same format. Refer to TRANSFORM() or @...SAY for more information on pictures.

<acColumnHeaders> is a parallel array of character expressions that define the headings for each column. Specifying <cColumnHeader> gives the same heading for all columns. To display a multiline heading, embed a semicolon in the heading expression where you want the string to break. If not specified, column headings are taken from the <acColumns> array, or the field names in the current work area if the <acColumns> argument is not specified.

<acHeadingSeparators> is a parallel array of character expressions that define the characters used to draw horizontal lines separating column headings from the field display area. Specifying <cHeadingSeparator> instead of an array uses the same heading separator for all columns. If this argument is not specified, the default separator is a double graphics line.

<acColumnSeparators> is a parallel array of character expressions that define the characters used to draw vertical lines separating the columns. Specifying <cColumnSeparator> instead of an array uses the same separator for all columns. If this argument is not specified, the default separator is a single graphics line.

<acFootingSeparators> is a parallel array of character expressions that define the characters used to draw horizontal lines separating column footings from the field display area. Specifying <cFootingSeparator> instead of an array uses the same footing separator for all columns. If this argument is not specified, there is no footing

separator.

<acColumnFootings> is a parallel array of character expressions that define footings for each column. Specifying <cColumnFooting> instead of an array gives the same footing for all columns. To display a multiline footing, embed a semicolon in the footing expression where you want the string to break. If this argument is not specified, there are no column footings.

Returns

DBEDIT() always returns NIL.

Description

DBEDIT() is a user interface and compatibility function that displays records from one or more work areas in a table form. The DBEDIT() window display is a grid of cells divided into columns and rows. Columns correspond to database fields and rows correspond to database records. Each column is defined by an element of the <acColumns> array. The display width of each column is determined by the evaluation of the column expression in <acColumns> array or the column picture specified in the <acColumnSayPictures> array.

All cursor movement keys are handled within DBEDIT(), including PgUp, PgDn, Home, End, the four arrow keys, and all Ctrl key combinations that produce cursor movement. The navigation keys that DBEDIT() responds to when a user function argument is not specified are listed in the Active Keys table below:

DBEDIT() Active Keys

Key	Action
Uparrow	Up one row
Dnarrow	Down one row
Leftarrow	Column left
Rightarrow	Column right
Ctrl-Leftarrow	Pan left one column
Ctrl-Rightarrow	Pan right one column
Home	Leftmost current screen column
End	Rightmost current screen column
Ctrl-Home	Leftmost column
Ctrl-End	Rightmost column
PgUp	Previous screen
PgDn	Next screen
Ctrl-PgUp	First row of current column
Ctrl-PgDn	Last row of current column
Return	Terminate DBEDIT()
Esc	Terminate DBEDIT()

When the user function argument (<cUserFunction>) is specified, all keys indicated in the Active Keys table are active with the exception of Esc and Return. When DBEDIT() calls the user function, it automatically passes two arguments:

. The current mode passed as a numeric value

. The index of the current column in <acColumns> passed as a numeric value

The mode parameter indicates the current state of DBEDIT() depending on the last key executed. The possible mode values are listed in the DBEDIT() Modes table below:

DBEDIT() Modes

Status	Dbedit.ch	Description
0	DE_IDLE	Idle, any cursor movement keystrokes have been handled and no keystrokes are pending
1	DE_HITTOP	Attempt to cursor past top of file
2	DE_HITBOTTOM	Attempt to cursor past bottom of file
3	DE_EMPTY	No records in work area
4	DE_EXCEPT	Key exception

The index parameter points to the position of the current column definition in the <acColumns> array. If <acColumns> is not specified, the index parameter points to the position of the field in the current database structure. Access the field name using FIELD().

A user-defined function must return a value that indicates to DBEDIT() the action to perform. The User Function Return Values table below lists the possible return values and the corresponding actions:

DBEDIT() User Function Return Values

Value	Dbedit.ch	Description
0	DE_ABORT	Abort DBEDIT()
1	DE_CONT	Continue DBEDIT()
2	DE_REFRESH	Force reread/repaint and continue; after repaint, process keys and go to idle

A number of instances effect calls to the user function:

. A key exception occurs. This happens when DBEDIT() fetches a keystroke that it does not recognize from the keyboard. Any pending keys remain in the keyboard buffer until fetched within the user function or until DBEDIT() continues.

. DBEDIT() enters the idle mode (i.e., all pending keys have been processed). This happens when the keyboard is empty or after a screen refresh. In this instance, there is one call to the user function and then DBEDIT() waits for a key.

. Beginning or end of file is encountered. This is the same as idle. All executable keys are performed, and there is one call to the user function with the appropriate status message.

Note that when DBEDIT() is first executed, all keys pending in the keyboard buffer are executed and then DBEDIT() enters the idle mode with a user function call. If no keys are pending, the idle mode is immediate.

The user function should handle all modes and status messages received from DBEDIT().

A user-defined function must assure that the DBEDIT() status is equivalent to DE_EXCEPT (4) otherwise the value of LASTKEY() is meaningless and a Return value of DE_REFRESH (2) will place the application into an endless loop.

```
FUNCTION DBEditFunc ( nMode, nColumnPos )
   LOCAL RetVal := DE_CONT

   IF ( nMode == DE_EXCEPT )
      IF ( LASTKEY() == K_F5 )
         RetVal := DE_REFRESH
      ENDIF
   ENDIF
RETURN(  RetVal )
```

DBEDIT() is fully reentrant, which means you can make nested calls to it. Using this feature, you can have multiple browse windows on the screen at the same time.

DBEDIT() is a compatibility function and, therefore, no longer recommended as a programmable browse facility. As such, it is superseded by the TBrowse object class. For more information, refer to TBrowse class in this chapter.

Examples

. This example demonstrates a generic call to DBEDIT():

```
USE Names NEW
DBEDIT()
```

. This example demonstrates calling DBEDIT() with a user function:

```
#include "dbedit.ch"
#include "inkey.ch"

// Array must be visible to other user-defined programs in
// program

STATIC acColumns := {}

PROCEDURE Main()

   USE Names NEW
   INDEX ON Names->Lastname + Names->FirstName TO Names

   CLS

   acColumns := { "LastName", "FirstName" }

   DBEDIT( 5, 5, 20, 70, acColumns, "UserFunc" )

RETURN


FUNCTION UserFunc( nMode, nCol )
   LOCAL nKey := LASTKEY()
   LOCAL nRetVal := DE_CONT         // Default return value

   DO CASE
   CASE nMode == DE_IDLE
      nRetVal := IdleFunc()
   CASE nMode == DE_HITTOP
      TONE( 100, 3 )
   CASE nMode == DE_HITBOTTOM
      TONE( 100, 3 )
      nRetVal := AppendFunc( nKey )
   CASE nMode == DE_EMPTY
      nRetVal := EmptyFunc()
   CASE nMode == DE_EXCEPT
      nRetVal := ExceptFunc( nKey, nCol )
   OTHERWISE
      TONE( 100, 3 )
   ENDCASE

RETURN nRetVal

FUNCTION AppendFunc( nKey )
   LOCAL nRetVal := DE_CONT         // Default return value

   IF nKey == K_DOWN                // If DOWN ARROW
      APPEND BLANK                  // Append blank record
```

```
// Note: The appended record will appear at the top of the
//       DBEDIT() screen when the database file is indexed.

      nRetVal := DE_REFRESH         // Refresh screen
   ENDIF


RETURN nRetVal

FUNCTION ExceptFunc( nKey, nCol )
   LOCAL nRetVal := DE_CONT         // Default return value

   DO CASE
   CASE nKey == K_ESC               // If ESCAPE
      nRetVal := DE_ABORT           // Exit
   CASE nKey == K_RETURN            // If RETURN
      nRetVal := EditFunc( nCol )   // Function to edit
                                    // field

   // Toggle DELETED status
   CASE nKey == K_DEL .AND. LASTREC() != 0  // DELETE pressed
      IF DELETED()
         RECALL
      ELSE
         DELETE
      ENDIF
   OTHERWISE
         TONE( 100, 1 )
   ENDCASE

RETURN nRetVal


FUNCTION EditFunc( nCol )
   LOCAL cIndexVal         // Value of current key expression
   LOCAL nRetVal           // Return value
   LOCAL nField            // Position of current field
   LOCAL cFieldVal         // Value of current field
   LOCAL nCursSave         // Preserve state of cursor

   // This will return an error if no index is open
   cIndexVal := &( INDEXKEY(0) )

   nField := FIELDPOS( acColumns[nCol] )

   IF nField != 0
      nCursSave := SETCURSOR()         // Save state of cursor
      SETCURSOR(1)                     // Change cursor shape
      cFieldVal := FIELDGET( nField )  // Save contents
                                       // of field
      @ ROW(), COL() GET cFieldVal     // GET new value
      READ
      FIELDPUT( nField, cFieldVal )    // REPLACE with
                                       // new value
      SETCURSOR( nCursSave )           // Restore cursor
                                       // shape
   ENDIF


   IF cIndexVal != &( INDEXKEY(0) )    // If key expression
                                       // changed

      nRequest := DE_REFRESH           // Refresh screen
   ELSE                                // Otherwise
      nRequest := DE_CONT              // Continue
   ENDIF

RETURN nRequest
```

```
FUNCTION IdleFunc()
   // Idle routine
RETURN DE_CONT

FUNCTION EmptyFunc()
   // Empty Records routine
RETURN DE_CONT
```

Files: Library is EXTEND.LIB, header files are Dbedit.ch and Inkey.ch.

See Also: @...GET ACHOICE() BROWSE()* MEMOEDIT() TRANSFORM()

DBEVAL()
Evaluate a code block for each record matching a scope and condition

---

Syntax

```
DBEVAL(<bBlock>,
   [<bForCondition>],
   [<bWhileCondition>],
   [<nNextRecords>],
   [<nRecord>],
   [<lRest>]) --> NIL
```

Arguments

<bBlock> is a code block to execute for each record processed.

<bForCondition> is an optional condition specified as a code block that is evaluated for each record in the scope. It provides the same functionality as the FOR clause of record processing commands.

<bWhileCondition> is an optional condition specified as a code block that is evaluated for each record from the current record until the condition returns false (.F.). It provides the same functionality as the WHILE clause of record processing commands.

<nNextRecords> is an optional number that specifies the number of records to process starting with the current record. It is the same as the NEXT clause.

<nRecord> is an optional record number to process. If this argument is specified, <bBlock> will be evaluated for the specified record. This argument is the same as the RECORD clause.

<lRest> is an optional logical value that determines whether the scope of DBEVAL() is all records, or, starting with the current record, all records to the end of file. This argument corresponds to the REST and ALL clauses of record processing commands. If true (.T.) , the scope is REST; otherwise, the scope is ALL records. If <lRest> is not specified the scope defaults to ALL.

Returns

DBEVAL() always returns NIL.

Description

DBEVAL() is a database function that evaluates a single block for each record within the current work area that matches a specified scope and/or condition. On each iteration, DBEVAL() evaluates the specified block. All records within the scope or matching the condition are processed until the end of file is reached.

By default, DBEVAL() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression.

DBEVAL() is similar to AEVAL() which applies a block to each element in an array. Like AEVAL(), DBEVAL() can be used as a primitive for the construction of user-defined commands that process database files. In fact, many of the standard CA-Clipper database processing commands are created using DBEVAL().

Refer to the Code Blocks section in the Basic Concepts chapter of the Programming and Utilities guide for more information on the syntax and theory of code blocks; refer also to the Database System section in the Basic Concepts chapter for information on record scoping and conditions; and refer to the CA-Clipper standard header file, STD.CH, found in

\CLIPPER5\INCLUDE for examples of CA-Clipper database command definitions that use DBEVAL().

Examples

. This example uses DBEVAL() to implement Count(), a user-defined function that counts the number of records in a work area matching a specified scope. The scope is passed as an array to Count(). To make the example more interesting, there is a user-defined command to create the scope array, thereby allowing you to specify the scope in a familiar form. Additionally, there is a set of manifest constants that define the attributes of the scope object:

```
// Scope command definition
#command CREATE SCOPE <aScope> [FOR <for>] ;
   [WHILE <while>] [NEXT <next>] [RECORD <rec>] ;
   [<rest:REST>] [ALL];
=>;
   <aScope> := { <{for}>, <{while}>, <next>, ;
      <rec>, <.rest.> }
//

// Scope attribute constants
#define FOR_COND     1
#define WHILE_COND   2
#define NEXT_SCOPE   3
#define REC_SCOPE    4
#define REST_SCOPE   5
//
// Create a scope and count records using it
LOCAL mySet, myCount
USE Customer NEW
CREATE SCOPE mySet FOR Customer = "Smith" WHILE ;
          Zip > "90000"
myCount := Count( mySet )
RETURN

FUNCTION Count( aScope )
   LOCAL nCount := 0
   DBEVAL( {|| nCount++},;
      aScope[ FOR_COND ],;
      aScope[ WHILE_COND ],;
      aScope[ NEXT_SCOPE ],;
      aScope[ REC_SCOPE ],;
      aScope[ REST_SCOPE ];
   )
   RETURN nCount
```

Files: Library is CLIPPER.LIB

See Also: AEVAL() EVAL()

DBF()*
Return current alias name

Syntax

DBF() --> cAlias

Returns

DBF() returns the alias of the current work area as a character string. If there is no active database file in the current work area, DBF() returns a null string ("").

Description

DBF() is a compatibility function that replicates the DBF() function in dBASE III PLUS. CA-Clipper implements it by invoking the ALIAS() function without an argument.

DBF() is a compatibility function and, therefore, no longer recommended. It is superseded entirely by the ALIAS() function.

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\DBF.PRG.

See Also: ALIAS() USED()

DBFILTER()
Return the current filter expression as a character string

Syntax

DBFILTER() --> cFilter

Returns

DBFILTER() returns the filter condition defined in the current work area as a character string. If no FILTER has been SET, DBFILTER() returns a null string ("").

Description

DBFILTER() is a database function used to save and reexecute an active filter by returning the filter expression as a character string that can be later recompiled and executed using the macro operator (&). This function operates like the DBRELATION() and DBRSELECT() functions which save and reexecute the linking expression of a relation within a work area.

Since each work area can have an active filter, DBFILTER() can return the filter expression of any work area. This is done by referring to DBFILTER() within an aliased expression as demonstrated below.

Notes

- **Declared variables:** A character string returned by DBFILTER() may not operate correctly when recompiled and executed using the macro operator (&) if the original filter expression contained references to local or static variables, or otherwise depended on compile-time declarations.

Examples

- This example opens two database files, sets two filters, then displays the filter expressions for both work areas:

```
USE Customer INDEX Customer NEW
SET FILTER TO Last = "Smith"
USE Invoices INDEX Invoices NEW
SET FILTER TO CustId = "Smi001"
SELECT Customer
//
? DBFILTER()                  // Result: Last = "Smith"
? Invoices->(DBFILTER())      // Result: Custid = "Smi001"
```

- This user-defined function, CreateQry(), uses DBFILTER() to create a memory file containing the current filter expression in the private variable cFilter:

```
FUNCTION CreateQry( cQryName )

   PRIVATE cFilter := DBFILTER()
   SAVE ALL LIKE cFilter TO (cQryName + ".qwy")
   RETURN NIL
```

- You can later RESTORE a query file with this user-defined function, SetFilter():

```
FUNCTION SetFilter()
PARAMETER cQryName
   RESTORE FROM &cQryName..qwy ADDITIVE
   SET FILTER TO &cFilter.
   RETURN NIL
```

Files: Library is CLIPPER.LIB.

See Also: DBRELATION() DBRSELECT() SET FILTER

DBGOBOTTOM()
Move to the last logical record

Syntax

DBGOBOTTOM() --> NIL

Returns

DBGOBOTTOM() always returns NIL.

Description

DBGOBOTTOM() moves to last logical record in the current work area.

DBGOBOTTOM() performs the same function as the standard GO BOTTOM command. For more information, refer to the GO command.

Notes

- **Logical records:** DBGOBOTTOM() operates on logical records. If there is an active index, DBGOBOTTOM() moves to the last record in indexed order. If a filter is set, only records which meet the filter condition are considered.

- **Controlling order:** If more than one index is active in the work area, the operation is performed using the controlling order as set by DBSETORDER() or the SET ORDER command. For more information, refer to the SET ORDER command.

- **Network environment:** For a shared file on a network, moving to a different record may cause updates to the current record to become visible to other processes. For more information, refer to the Network Programming chapter in the Programming and Utilities guide.

Examples

- The following example uses DBGOBOTTOM() to position the record pointer on the last logical record:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "LASTNAME" )
//
Sales->( DBGOBOTTOM() )
IF ( Sales->Last == "Winston" )
   IF RLOCK()
      Sales->( DBDELETE() )
      ? "Record deleted: ", Sales( DELETED() )
   ELSE
      ? "Unable to lock record..."
   ENDIF
END
```

Files: Library is CLIPPER.LIB.

See Also: BOF() DBGOTOP() DBSEEK() DBSKIP() EOF() GO

DBGOTO()
Move to the record having the specified record number

Syntax

DBGOTO(<nRecordNumber>) --> NIL

Arguments

<nRecordNumber> is a numeric value that specifies the record number of the desired record.

Returns

DBGOTO() always returns NIL.

Description

DBGOTO() moves to the record whose record number is equal to <nRecordNumber>. If no such record exists, the work area is positioned to LASTREC() + 1 and both EOF() and BOF() return true (.T.).

DBGOTO() performs the same function as the standard GO command. For more information, refer to the GO command.

Notes

- **Logical records:** DBGOTO() does not respect logical visibility. That is, if the specified record exists, it will become the current record regardless of any index or filter condition.

- **Network environment:** For a shared file on a network, moving to a different record may cause updates to the current record to become visible to other processes. For more information, refer to the Network Programming chapter in the Programming and Utilities guide.

Examples

- The following example uses DBGOTO() to iteratively process every fourth record:

```
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
//
// toggle every fourth record
DO WHILE !EOF()
   DBGOTO( RECNO() + 4 )
   Sales->Group := "Bear"
ENDDO
```

Files: Library is CLIPPER.LIB.

See Also: BOF() DBGOBOTTOM() DBGOTOP() DBSEEK() DBSKIP() EOF() GO

DBGOTOP()
Move to the first logical record

Syntax

DBGOTOP() --> NIL

Returns

DBGOTOP() always returns NIL.

Description

DBGOTOP() moves to the first logical record in the current work area.

DBGOTOP() performs the same function as the standard GO TOP command. For more information, refer to the GO TOP command.

Notes

- **Logical records:** DBGOTOP() operates on logical records. If there is an active index, DBGOTOP() moves to the first record in indexed order. If a filter is set, only records which meet the filter condition are considered.

- **Controlling order:** If more than one index is active in the work area, the operation is performed using the controlling order as set by DBSETORDER() or the SET ORDER command. For more information, refer to the SET ORDER command.

- **Network environment:** For a shared file on a network, moving to a different record may cause updates to the current record to become visible to other processes. For more information, refer to the Network Programming chapter in the Programming and Utilities guide.

Examples

- This example demonstrates the typical use of DBGOTOP():

```
DBGOTOP()
WHILE ( !EOF() )
   ? FIELD->Name
   DBSKIP()
END
```

Files: Library is CLIPPER.LIB.

See Also: BOF() DBGOBOTTOM() DBSEEK() DBSKIP() EOF() GO

DBRECALL()
Reinstate a record marked for deletion

Syntax

DBRECALL() --> NIL

Returns

DBRECALL() always returns NIL.

Description

DBRECALL() causes the current record to be reinstated if it is marked for deletion.

DBRECALL() performs the same function as the RECALL command. For more information, refer to the DELETE and RECALL commands.

Notes

- **Logical records:** Reinstating a deleted record affects the record's logical visibility if the global _SET_DELETED status is true (.T.). For more information, refer to the DBDELETE() function and the DELETE and RECALL commands.

- **Network environment:** For a shared database on a network, DBRECALL() requires the current record to be locked. For more information, refer to the Network Programming chapter in the Programming and Utilities guide.

Examples

- The following example recalls a record if it is deleted and attempts to lock the record if successful:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "LASTNAME" )
//
IF ( Sales->(DBSEEK(cLast)) )
   IF Sales->( DELETED() )

      IF Sales( RLOCK() )
         Sales( DBRECALL() )
         ? "Record recalled"
      ELSE
         ? "Unable to lock record..."
      ENDIF
   ENDIF
ELSE
   ? "Not found"
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: DBDELETE() DELETE RECALL

DBREINDEX()
Recreate all active indexes for the current work area

Syntax

DBREINDEX() --> NIL

Returns

DBREINDEX() always returns NIL.

Description

DBREINDEX() rebuilds all active indexes associated with the current work area. After the indexes are recreated, the work area is moved to the first logical record in the controlling order.

DBREINDEX() performs the same function as the standard REINDEX command. For more information, refer to the REINDEX command.

Examples

. The following example reindexes the work area:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "LASTNAME" )
//
IF ( Sales->(DBSEEK(cLast)) )
   IF RLOCK()
      DELETE FOR Sales->LastName == "Winston"
      Sales->( DBREINDEX() )
   ELSE
      ? "Unable to lock record..."
   ENDIF
ELSE
   ? "Not found"
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: DBCLEARIND() DBCREATEIND() DBSETINDEX() DBSETORDER() REINDEX

DBRELATION()
Return the linking expression of a specified relation

Syntax

DBRELATION(<nRelation>) --> cLinkExp

Arguments

<nRelation> is the position of the desired relation in the list of current work area relations. The relations are numbered according to the order in which they were defined with SET RELATION.

Returns

DBRELATION() returns a character string containing the linking expression of the relation specified by <nRelation>. If there is no RELATION SET for <nRelation>, DBRELATION() returns a null string ("").

Description

DBRELATION() is a database function used with DBRSELECT() to determine the linking expression and work area of an existing relation created with the SET RELATION command.

DBRELATION() returns the linking expression defined by the TO clause. DBRSELECT() returns the work area linked as defined by the INTO clause.

By default, DBRELATION() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

Notes

- **Declared variables:** A character string returned by DBRELATION() may not operate correctly when recompiled and executed using the macro operator (&) if the original expression contained references to local or static variables, or otherwise depends on compile-time declarations.

Examples

- This example opens three database files, sets two child relations from the parent work area, and then displays the linking expression to the second child work area:

```
USE Invoices INDEX Invoices NEW
USE BackOrder INDEX BackOrder NEW
USE Customer INDEX Customer NEW
SET RELATION TO CustNum INTO Invoices, OrderNum ;
      INTO BackOrder
//

? DBRELATION(2)               // Result: OrderNum
```

- Later you can query the same linking expression from an unselected work area by using an aliased expression like this:

```
USE Archive NEW
? Customer->(DBRELATION(2))   // Result: OrderNum
```

- This example is a user-defined function, Relation(), that returns the results of both DBRELATION() and DBRSELECT() as an array:

```
FUNCTION Relation( nRelation )
   RETURN { DBRELATION(nRelation), ;
         ALIAS(DBRSELECT(nRelation)) }
```

Files: Library is CLIPPER.LIB.

See Also: DBFILTER() DBRSELECT() SET RELATION

DBRSELECT()
Return the target work area number of a relation

Syntax

DBRSELECT(<nRelation>) --> nWorkArea

Arguments

<nRelation> is the position of the desired relation in the list of current work area relations. The relations are numbered according to the order in which they were defined with SET RELATION.

Returns

DBRSELECT() returns the work area number of the relation specified by <nRelation> as an integer numeric value. If there is no RELATION SET for <nRelation>, DBRSELECT() returns zero.

Description

DBRSELECT() is a database function used in combination with DBRELATION() to determine the work area and linking expression of an existing relation created with the SET RELATION command. DBRSELECT() returns the work area defined by the INTO clause. DBRELATION() returns the linking expression defined by the TO clause. To determine the alias of the relation instead of the work area number, use the expression ALIAS(DBRSELECT(<nRelation>)).

By default, DBRSELECT() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

Examples

. This example opens three database files, sets two child relations from the parent work area, then displays the linking expression to the second child work area as well as the target work area of the relation:

```
USE Invoices INDEX Invoices NEW
USE BackOrder INDEX BackOrder NEW
USE Customer INDEX Customer NEW
SET RELATION TO CustNum INTO Customer, ;
      OrderNum INTO BackOrder
//
? DBRELATION(2), DBRSELECT(2)    // Result: OrderNum 3
? ALIAS(DBRSELECT(2))            // Result: BACKORDER
```

. Later, you can query the same information from an unselected work area by using an aliased expression:

```
USE Archive NEW
? Customer->(DBRELATION(2))      // Result: OrderNum
? Customer->(DBRSELECT(2))       // Result: 3
```

Files: Library is CLIPPER.LIB.

See Also: DBFILTER() DBRELATION() SET RELATION

DBSEEK()
Move to the record having the specified key value

Syntax

DBSEEK(<expKey>, [<lSoftSeek>]) --> lFound

Arguments

<expKey> is a value of any type that specifies the key value associated with the desired record.

<lSoftSeek> is an optional logical value that specifies whether a soft seek is to be performed. This determines how the work area is positioned if the specified key value is not found (see below). If <lSoftSeek> is omitted, the current global _SET_SOFTSEEK setting is used.

Returns

DBSEEK() returns true (.T.) if the specified key value was found; otherwise, it returns false (.F.).

Description

DBSEEK() moves to the first logical record whose key value is equal to <expKey>. If such a record is found, it becomes the current record and DBSEEK() returns true (.T.). Otherwise, DBSEEK() returns false (.F.) and the positioning of the work area is as follows: for a normal (not soft) seek, the work area is positioned to LASTREC() + 1 and EOF() returns true (.T.); for a soft seek, the work area is positioned to the first record whose key value is greater than the specified key value. If no such record exists, the work area is positioned to LASTREC() + 1 and EOF() returns true (.T.).

For a work area with no active indexes, DBSEEK() has no effect.

DBSEEK() performs the same function as the standard SEEK command. For more information, refer to the SEEK command.

Notes

- **Logical records:** DBSEEK() operates on logical records. Records are considered in indexed order. If a filter is set, only records which meet the filter condition are considered.

- **Controlling order:** If the work area has more than one active index, the operation is performed using the controlling order as set by DBSETORDER() or the SET ORDER command. For more information, refer to the SET ORDER command.

- **Network environment:** For a shared file on a network, moving to a different record may cause updates to the current record to become visible to other processes. For more information, refer to the Network Programming chapter in the Programming and Utilities guide.

Examples

- In this example, DBSEEK() moves the pointer to the record in the database, Employee, in which the value in FIELD .cName. matches the entered value of cName:

```
ACCEPT "Employee name: " TO cName
IF ( Employee->(DBSEEK(cName)) )
   Employee->(ViewRecord())
ELSE
   ? "Not found"
```

```
        END
```

Files: Library is CLIPPER.LIB.

See Also: DBGOBOTTOM() DBGOTOP() DBSKIP() EOF() FOUND() SEEK

DBSELECTAREA()
Change the current work area

Syntax

DBSELECTAREA(<nArea> | <cAlias>) --> NIL

Arguments

<nArea> is a numeric value between zero and 250, inclusive, that specifies the work area being selected.

<cAlias> is a character value that specifies the alias of a currently occupied work area being selected.

Returns

DBSELECTAREA() always returns NIL.

Description

DBSELECTAREA() causes the specified work area to become the current work area. All subsequent database operations will apply to this work area unless another work area is explicitly specified for an operation. DBSELECTAREA() performs the same function as the standard SELECT command. For more information, refer to the SELECT command.

Notes

- **Selecting zero:** Selecting work area zero causes the lowest numbered unoccupied work area to become the current work area.

- **Aliased expressions:** The alias operator (->) can temporarily select a work area while an expression is evaluated and automatically restore the previously selected work area afterward. For more information, refer to the alias operator (->).

Examples

- The following example selects a work area via the alias name:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "SALEFNAM" )
DBSETINDEX( "SALELNAM" )
//
DBUSEAREA( .T., "DBFNTX", "Colls", "Colls", .T. )
DBSETINDEX( "COLLFNAM" )
DBSETINDEX( "COLLLNAM" )
//
DBSELECTAREA( "Sales" )      // select "Sales" work area
//
IF ( Sales->(DBSEEK(cLast)) )
   IF Sales->( DELETED() ) .AND. Sales->( RLOCK() )
      Sales->( DBRECALL() )
      ? "Deleted record has been recalled."
   ENDIF
ELSE
   ? "Not found"
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: DBUSEAREA() SELECT SELECT() USE

DBSETDRIVER()
Return the default database driver and optionally set a new driver

---

Syntax

DBSETDRIVER([<cDriver>]) --> cCurrentDriver

Arguments

<cDriver> is an optional character value that specifies the name of the database driver that should be used to activate and manage new work areas when no driver is explicitly specified.

Returns

DBSETDRIVER() returns the name of the current default driver.

Description

DBSETDRIVER() sets the database driver to be used when activating new work areas without specifying a driver. If the specified driver is not available to the application, the call has no effect. DBSETDRIVER() returns the name of the current default driver, if any.

Examples

. This example makes the "DBFNDX" driver the default driver. If the driver is unavailable, a message is issued:

```
DBSETDRIVER("DBFNDX")
IF ( DBSETDRIVER() <> "DBFNDX" )
   ? "DBFNDX driver not available"
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: DBUSEAREA() USE

DBSETFILTER()
Set a filter condition

Syntax

DBSETFILTER(<bCondition>, [<cCondition>]) --> NIL

Arguments

<bCondition> is a code block that expresses the filter condition in executable form.

<cCondition> is an optional character value that expresses the filter condition in textual form. If <cCondition> is omitted, the DBSETFILTER() function will return an empty string for the work area.

Returns

DBSETFILTER() always returns NIL.

Description

DBSETFILTER() sets a logical filter condition for the current work area. When a filter is set, records which do not meet the filter condition are not logically visible. That is, database operations which act on logical records will not consider these records.

The filter expression supplied to DBSETFILTER() evaluates to true (.T.) if the current record meets the filter condition; otherwise, it should evaluate to false (.F.).

The filter expression may be a code block (<bCondition>) or both a code block and equivalent text (<cCondition>). If both versions are supplied, they must express the same condition. If the text version is omitted, DBFILTER() will return an empty string for the work area.

DBSETFILTER() performs the same function as the standard SET FILTER command. For more information, refer to the SET FILTER command.

Notes

- **Logical records:** DBSETFILTER() affects the logical visibility of records (see above).

- **Side effects:** Setting a filter condition is only guaranteed to restrict visibility of certain records as described above. The filter expression is not necessarily evaluated at any particular time, by any particular means, or on any particular record or series of records. If the filter expression relies on information external to the database file or work area, the effect is unpredictable. If the filter expression changes the state of the work area (e.g., by moving to a different record or changing the contents of a record), the effect is unpredictable.

- **Evaluation context:** When the filter expression is evaluated, the associated work area is automatically selected as the current work area before the evaluation; the previously selected work area is automatically restored afterward.

Examples

- This example limits data access to records in which the Age field value is less than 40:

```
USE Employee NEW
DBSETFILTER( {|| Age < 40}, "Age < 40" )
DBGOTOP()
```

Files: Library is CLIPPER.LIB.

See Also: DBCLEARFIL() DBFILTER() SET DELETED SET FILTER

DBSETINDEX()
Open an index

Syntax

DBSETINDEX(<cIndexName>) --> NIL

Arguments

<cIndexName> is a character value that specifies the filename of the index file to be opened.

Returns

DBSETINDEX() always returns NIL.

Description

DBSETINDEX() opens an index file for the current work area. Any index files already associated with the work area continue to be active. If the newly opened index is the only index associated with the work area, it becomes the controlling index; otherwise, the controlling order remains unchanged.

After the new index file is opened, the work area is positioned to the first logical record in the controlling order.

DBSETINDEX() performs the same function as the standard SET INDEX command or the INDEX clause of the USE command. For more information, refer to the SET INDEX command.

Notes

- **Logical records:** Activating an index affects the logical order of records in the work area. That is, when an index is active, commands which operate on logical records process records in indexed order. If the controlling index is a unique index, records with duplicate index keys are not logically visible.

- **Network environment:** DBSETINDEX() opens the index for shared access if the associated database file was opened for shared access. Otherwise, the index opens for exclusive access.

Examples

- The following example sets multiple indexes:

```
cLast := "Winston"
DBUSEAREA( .T., "DBFNTX", "Sales", "Sales", .T. )
DBSETINDEX( "FIRSTNAM" )
DBSETINDEX( "LASTNAME" )
//
IF ( Sales->(DBSEEK(cLast)) )
   IF Sales->( DELETED() ) .AND. Sales->( RLOCK() )

      Sales->( DBRECALL() )
      ? "Deleted record has been recalled."
   ENDIF
ELSE
   ? "Not found"
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: DBCLEARIND() DBCREATEIND() DBREINDEX() DBSETORDER() SET INDEX

DBSETORDER()
Set the controlling order

Syntax

DBSETORDER(<nOrderNum>) --> NIL

Arguments

<nOrderNum> is a numeric value that specifies which of the active indexes is to be the controlling index.

Returns

DBSETORDER() always returns NIL.

Description

DBSETORDER() controls which of the current work area's active indexes is the controlling index. The controlling index is the index which determines the logical order of records in the work area.

Active indexes are numbered from 1 to the number of active indexes, based on the order in which the indexes were opened. <nOrderNum> specifies the number of the desired index.

DBSETORDER() performs the same function as the standard SET ORDER command. For more information, refer to the SET ORDER command.

Notes

- **Setting order to zero:** Setting order to zero causes the work area to be accessed in natural (record number) order. Only the logical order of the records is affected; any open indexes continue to be active and are properly maintained.

Examples

- This example sets the second named index, Age, as the controlling index:

```
USE Employee NEW
SET INDEX TO Name, Age

DBSETORDER(2)
```

Files: Library is CLIPPER.LIB.

See Also: DBCLEARIND() DBCREATEIND() DBREINDEX() DBSETINDEX() SET ORDER

DBSETRELATION()
Relate two work areas

Syntax

DBSETRELATION(<nArea> | <cAlias>, <bExpr>, [<cExpr>])
--> NIL

Arguments

<nArea> is a numeric value that specifies the work area number of the child work area.

<cAlias> is a character value that specifies the alias of the child work area.

<bExpr> is a code block that expresses the relational expression in executable form.

<cExpr> is an optional character value that expresses the relational expression in textual form. If <cExpr> is omitted, the DBRELATION() function returns an empty string for the relation.

Returns

DBSETRELATION() always returns NIL.

Description

DBSETRELATION() relates the work area specified by <nArea> or <cAlias> (the child work area), to the current work area (the parent work area). Any existing relations remain active.

Relating work areas synchronizes the child work area with the parent work area. This is achieved by automatically repositioning the child work area whenever the parent work area moves to a new record. If there is an active index in the child work area, moving the parent work area causes an automatic SEEK operation in the child work area; the seek key is based on the expression specified by <bExpr> and/or <cExpr>. If the child work area has no active index, moving the parent work area causes an automatic GOTO in the child work area; the record number for the GOTO is based on the expression specified by <bExpr> and/or <cExpr>.

The relational expression may be a code block (<bExpr>) or both a code block and equivalent text (<cExpr>). If both versions are supplied, they must be equivalent. If the text version is omitted, DBRELATION() will return an empty string for the relation.

DBSETRELATION() performs the same function as the standard SET RELATION command with the ADDITIVE clause. For more information, refer to the SET RELATION command.

Notes

. **Side effects:** DBSETRELATION() is only guaranteed to synchronize the work areas as described above. The relational expression is not necessarily evaluated at any particular time, by any particular means, or on any particular record or series of records. If the relational expression relies on information external to the parent work area or its associated database file, the effect is unpredictable. If the expression changes the state of either work area (e.g., by moving to a different record or changing the contents of a record), the effect is unpredictable.

. **Evaluation context:** When the relational expression is evaluated, the parent work area is automatically selected as the current work area before the evaluation; the previously selected work

area is automatically restored afterward.

- **Soft seeking:** Seek operations that occur as part of relational positioning are never soft seeks. If a relational movement is unsuccessful, the child work area is positioned to LASTREC() + 1, its FOUND() status returns false (.F.), and its EOF() status returns true (.T.).

Examples

- This example demonstrates a typical use of the DBSETRELATION() function:

```
USE Employee NEW
USE Department NEW INDEX Dept
SELECT Employee
DBSETRELATION("Department", {|| Employee->Dept},;
                "Employee->Dept")
LIST Employee->Name, Department->Name
```

Files: Library is CLIPPER.LIB.

See Also: DBCLEARREL() DBRELATION() DBRSELECT() FOUND() SET RELATION

DBSKIP()
Move relative to the current record

Syntax

DBSKIP([<nRecords>]) --> NIL

Arguments

<nRecords> is the number of logical records to move, relative to the current record. A positive value means to skip forward, and a negative value means to skip backward. If <nRecords> is omitted, a value of 1 is assumed.

Returns

DBSKIP() always returns NIL.

Description

DBSKIP() moves either forward or backward relative to the current record. Attempting to skip forward beyond the last record positions the work area to LASTREC() + 1 and EOF() returns true (.T.). Attempting to skip backward beyond the first record positions the work area to the first record and BOF() returns true (.T.).

DBSKIP() performs the same function as the standard SKIP command. For more information, refer to the SKIP command.

Notes

- **Logical records:** DBSKIP() operates on logical records. If there is an active index, records are considered in indexed order. If a filter is set, only records which meet the filter condition are considered.

- **Controlling order:** If the work area has more than one active index, the skip operation is performed using the controlling order as set by DBSETORDER() or the SET ORDER command. For more information, refer to the SET ORDER command.

- **Network environment:** For a shared file on a network, moving to a different record may cause updates to the current record to become visible to other processes. For more information, refer to the Network Programming chapter in the Programming and Utilities guide.

Examples

- This example demonstrates a typical use of the DBSKIP() function:

```
DBGOTOP()
DO WHILE ( !EOF() )
   ? FIELD->Name
   DBSKIP()
ENDDO
```

Files: Library is CLIPPER.LIB.

See Also: BOF() DBGOBOTTOM() DBGOTOP() DBSEEK() EOF() SKIP

DBSTRUCT()
Create an array containing the structure of a database file

Syntax

DBSTRUCT() --> aStruct

Returns

DBSTRUCT() returns the structure of the current database file in an array whose length is equal to the number of fields in the database file. Each element of the array is a subarray containing information for one field. The subarrays have the following format:

DBSTRUCT() Return Array

Position	Metasymbol	Dbstruct.ch
1	cName	DBS_NAME
2	cType	DBS_TYPE
3	nLength	DBS_LEN
4	nDecimals	DBS_DEC

If there is no database file in USE in the current work area, DBSTRUCT() returns an empty array ({}).

Description

DBSTRUCT() is a database function that operates like COPY STRUCTURE EXTENDED by creating an array of structure information rather than a database file of structure information. There is another function, DBCREATE(), that can create a database file from the structure array.

By default, DBSTRUCT() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression as shown below.

Note, a header file, Dbstruct.ch, located in \CLIPPER5\INCLUDE contains a series of manifest constants for each field attribute.

Examples

. This example opens two database files then creates an array containing the database structure using DBSTRUCT() within an aliased expression. The field names are then listed using AEVAL():

```
#include "Dbstruct.ch"
//
LOCAL aStruct
USE Customer NEW
USE Invoices NEW
//
aStruct := Customer->(DBSTRUCT())
AEVAL( aStruct, {|aField| QOUT(aField[DBS_NAME])} )
```

Files: Library is CLIPPER.LIB, header file is Dbstruct.ch.

See Also: AFIELDS()* COPY STRU EXTE

DBUNLOCK()
Release all locks for the current work area

Syntax

DBUNLOCK() --> NIL

Returns

DBUNLOCK() always returns NIL.

Description

DBUNLOCK() releases any record or file locks obtained by the current process for the current work area. DBUNLOCK() is only meaningful on a shared database in a network environment.

DBUNLOCK() performs the same function as the standard UNLOCK command. For more information, refer to the UNLOCK command.

Notes

- **Network environment:** Releasing locks may cause updates to the database to become visible to other processes. For more information, refer to the Network Programming chapter in the Programming and Utilities guide.

Examples

- The following example illustrates a basic use of the DBUNLOCK() function:

```
cLast := "Winston"
USE Sales SHARED NEW VIA "DBFNTX"
DBSETINDEX( "LASTNAME" )
//
IF ( Sales->(DBSEEK(cLast)) )
   IF Sales->( RLOCK() )
      Sales->( DBDELETE() )
      ? "Record deleted: ", Sales( DELETED() )
      Sales->( DBUNLOCK() )
   ELSE
      ? "Unable to lock record..."
   ENDIF
ELSE
   ? "Not found"
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: DBUNLOCKALL() FLOCK() RLOCK() UNLOCK

DBUNLOCKALL()
Release all locks for all work areas

Syntax

DBUNLOCKALL() --> NIL

Returns

DBUNLOCKALL() always returns NIL.

Description

DBUNLOCKALL() releases any record or file locks obtained by the current process for any work area. DBUNLOCKALL() is only meaningful on a shared database in a network environment. It is equivalent to calling DBUNLOCK() on every occupied work area.

DBUNLOCKALL() performs the same function as the UNLOCK ALL command. For more information, refer to the UNLOCK ALL command.

Examples

. The following example marks a record for deletion if an RLOCK() attempt is successful, then clears all locks in all work areas:

```
cLast := "Winston"
USE Sales SHARED NEW VIA "DBFNTX"
DBSETINDEX( "SALEFNAM" )
DBSETINDEX( "SALELNAM" )
//
USE Colls SHARED NEW VIA "DBFNTX"
DBSETINDEX( "COLLFNAM" )
DBSETINDEX( "COLLLNAM" )
//
DBSELECTAREA( "Sales" )     // select "Sales" work area
//
IF ( Colls->(DBSEEK(cLast)) )
   IF Colls->( DELETED() )
      ? "Record deleted: ", Colls->( DELETED() )
      IF Colls->( RLOCK() )
         Colls->( DBRECALL() )
         ? "Record recalled..."
      ENDIF
   ENDIF
ELSE
   ? "Not found"
   DBUNLOCKALL()            // remove all locks in
ENDIF                       // all work areas
```

Files: Library is CLIPPER.LIB.

See Also: DBUNLOCK() FLOCK() RLOCK() UNLOCK

DBUSEAREA()
Use a database file in a work area

Syntax

DBUSEAREA([<lNewArea>], [<cDriver>], <cName>, [<xcAlias>], [<lShared>], [<lReadonly>]) --> NIL

Arguments

<lNewArea> is an optional logical value. A value of true (.T.) selects the lowest numbered unoccupied work area as the current work area before the use operation. If <lNewArea> is false (.F.) or omitted, the current work area is used; if the work area is occupied, it is closed first.

<cDriver> is an optional character value. If present, it specifies the name of the database driver which will service the work area. If <cDriver> is omitted, the current default driver is used (see note below).

<cName> specifies the name of the database (.dbf) file to be opened.

<xcAlias> is an optional character value. If present, it specifies the alias to be associated with the work area. The alias must constitute a valid CA-Clipper identifier. A valid <xcAlias> may be any legal identifier (i.e., it must begin with an alphabetic character and may contain numeric or alphabetic characters and the underscore). Within a single application, CA-Clipper will not accept duplicate aliases. If <xcAlias> is omitted, a default alias is constructed from <cName>.

<lShared> is an optional logical value. If present, it specifies whether the database (.dbf) file should be accessible to other processes on a network. A value of true (.T.) specifies that other processes should be allowed access; a value of false (.F.) specifies that the current process is to have exclusive access. If <lShared> is omitted, the current global _SET_EXCLUSIVE setting determines whether shared access is allowed.

<lReadonly> is an optional logical value that specifies whether updates to the work area are prohibited. A value of true (.T.) prohibits updates; a value of false (.F.) permits updates. A value of true (.T.) also permits read-only access to the specified database (.dbf) file. If <lReadonly> is omitted, the default value is false (.F.).

Returns

DBUSEAREA() always returns NIL.

Description

DBUSEAREA() associates the specified database (.dbf) file with the current work area.

DBUSEAREA() performs the same function as the standard USE command. For more information, refer to the USE command.

Notes

. **Current driver:** If no driver is specified in the call to DBUSEAREA() the default driver is used. If more than one driver is available to the application, the default driver is the driver specified in the most recent call to DBSETDRIVER(). If DBSETDRIVER() has not been called, the name of the default driver is undetermined.

Examples

. This example is a typical use of the DBUSEAREA() function:

```
DBUSEAREA(.T., "DBFNDX", "Employees")
```

Files: Library is CLIPPER.LIB.

See Also: DBCLOSEAREA() DBSETDRIVER() SELECT() SET() USE

DELETED()
Return the deleted status of the current record

Syntax

DELETED() --> lDeleted

Returns

DELETED() returns true (.T.) if the current record is marked for deletion; otherwise, it returns false (.F.). If there is no database file in USE in the current work area, DELETED() returns false (.F.).

Description

DELETED() is a database function that determines if the current record in the active work area is marked for deletion. Since each work area with an open database file can have a current record, each work area has its own DELETED() value.

By default, DELETED() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

In applications, DELETED() is generally used to query the deleted status as a part of record processing conditions, or to display the deleted status as a part of screens and reports.

Examples

. This example uses DELETED() in the current and in an unselected work area:

```
USE Customer NEW

USE Sales NEW
? DELETED()                     // Result: .F.
DELETE
? DELETED()                     // Result: .T.
? Customer->(DELETED())         // Result: .F.
```

. This example uses DELETED() to display a record's deleted status in screens and reports:

```
@ 1, 65 SAY IF(DELETED(), "Inactive", "Active")
```

Files: Library is CLIPPER.LIB.

See Also: DELETE PACK RECALL SET DELETED

DESCEND()
Create a descending index key value

Syntax

DESCEND(<exp>) --> ValueInverted

Arguments

<exp> is any valid expression of character, date, logical, or numeric type. Memo type is treated in the same way as character type.

Returns

DESCEND() returns an inverted expression of the same data type as the <exp>, except for dates which return a numeric value. A DESCEND() of CHR(0) always returns CHR(0).

Description

DESCEND() is a conversion function that returns the inverted form of the specified expression to be used with INDEX to create descending order indexes. Specify that part of the index expression you want to be descending as the DESCEND() argument. To subsequently perform a lookup with SEEK, specify DESCEND() in the search expression.

Notes

- The preferred way to create a descending index is to use the DESCENDing clause of the INDEX command. Using DESCENDING is the same as specifying the DESCEND() function, but without the performance penalty during index updates. If you create a DESCENDING index, you won't need to use the DESCEND() function during a SEEK. DESCENDING is an attribute of the (.ntx) file, where it is stored and used for REINDEXing purposes.

Examples

- This example uses DESCEND() in an INDEX expression to create a descending order date index:

```
USE Sales NEW
INDEX ON DESCEND(OrdDate) TO SalesDate
```

Later, use DESCEND() to SEEK on the descending index:

```
SEEK DESCEND(dFindDate)
```

- This example illustrates how to create a descending order index using more than one data type. Here, the key is created using the concatenation of date and character fields after the appropriate type conversion has taken place. This example uses STR() instead of DTOS() since DESCEND() of a date returns a numeric value:

```
USE Sales NEW
INDEX ON STR(DESCEND(SaleDate)) + Salesman TO LastSale
```

Files: Library is EXTEND.LIB.

See Also: FIND* INDEX SEEK

DEVOUT()
Write a value to the current device

Syntax

DEVOUT(<exp>, [<cColorString>]) --> NIL

Arguments

<exp> is the value to display.

<cColorString> is an optional argument that defines the display color of <exp>. If the current DEVICE setting is SCREEN, the output is displayed in the specified color.

If not specified, <exp> is displayed as the standard color of the current system color as defined by SETCOLOR(). <cColorString> is a character expression containing the standard color setting. If you want to specify a literal color setting, enclose it in quote marks.

Returns

DEVOUT() always returns NIL.

Description

DEVOUT() is a full-screen display function that writes the value of a single expression to the current device at the current cursor or printhead position. Use DEVOUT() with DEVPOS() in STD.CH to implement the @...SAY command.

Examples

. This example shows the relationship between the DEVOUT() function and the @...SAY command:

```
DEVPOS(10, 10)
DEVOUT("Hello there", "BG+/B"))
//
@ 10, 10 SAY "Hello there" COLOR "BG+/B"
```

Files: Library is CLIPPER.LIB.

See Also: COL() DEVPOS() QOUT() ROW() SETPOS() Colors

DEVOUTPICT()
Write a value to the current device using a picture clause

Syntax

DEVOUTPICT(<exp>, <cPicture>, [<cColorString>])
--> NIL

Arguments

<exp> is the value to display.

<cPicture> defines the formatting control for the display of <exp>. The picture specified here is the same as one used with @...SAY or TRANSFORM and can include both templates and functions.

<cColorString> is an optional argument that defines the display color of <exp>. If the current DEVICE is SCREEN, output displays in the specified color.

If not specified, <exp> displays as the standard color of the current system color as defined by SETCOLOR(). <cColorString> is a character expression containing the standard color setting. If you want to specify a literal color setting, it must be enclosed in quote marks.

Returns

DEVOUTPICT() always returns NIL.

Description

DEVOUTPICT() is a full-screen display function that writes the value of a single expression to the current device at the current cursor or printhead position. DEVOUTPICT() is used in combination with DEVPOS() in STD.CH to implement the @...SAY command used with a PICTURE clause.

Examples

. This example shows the relationship between the DEVOUTPICT() function and the @...SAY command:

```
DEVPOS(10, 10)

DEVOUTPICT("Hello there", "@!", "BG+/B"))
//
@ 10, 10 SAY "Hello there" PICTURE "@!" COLOR "BG+/B"
```

Files: Library is CLIPPER.LIB.

See Also: @...SAY DEVOUT() DEVPOS() SETPOS() TRANSFORM() Colors

DEVPOS()
Move the cursor or printhead to a new position depending on the current device

---

Syntax

DEVPOS(<nRow>, <nCol>) --> NIL

Arguments

<nRow> and **<nCol>** are the new row and column positions of the cursor or printhead.

Returns

DEVPOS() always returns NIL.

Description

DEVPOS() is an environment function that moves the screen or printhead depending on the current DEVICE. If DEVICE is SET to SCREEN, DEVPOS() behaves like SETPOS(), moves the cursor to the specified location, and updates ROW() and COL() with the new cursor position.

If DEVICE is SET to PRINTER, DEVPOS() moves the printhead instead. It does this by sending the number of linefeed and/or formfeed characters to the printer, advancing the printhead to the new position. If the current SET MARGIN value is greater than zero, it is added to <nCol>. The printhead is then advanced to the specified <nRow> and <nCol> position and PROW() and PCOL() are updated. If either <nRow> or <nCol> are less than the current PROW() and PCOL() values, the printhead is moved according to the following special rules:

. If <nRow> is less than PROW(), an automatic EJECT (CHR(12)) is sent to the printer followed by the number of linefeed characters (CHR(10)) required to position the printhead on <nRow> of the following page.

. If <nCol> including the current SET MARGIN value is less than PCOL(), a carriage return character (CHR(13)) and the number of space characters required to position the printhead at <nCol> are sent to the printer.

To circumvent these rules, use SETPRC() to reset PROW() and PCOL() to new values before using DEVPOS(). See the SETPRC() discussion for more information.

If the printer is redirected to a file using the SET PRINTER command, DEVPOS() updates the file instead of the printer.

Files: Library is CLIPPER.LIB.

See Also: @...SAY DEVOUT() PCOL() PROW() SET DEVICE SETPOS()

DIRECTORY()
Create an array of directory and file information

Syntax

DIRECTORY(<cDirSpec>, [<cAttributes>]) --> aDirectory

Arguments

<cDirSpec> identifies the drive, directory and file specification for the directory search. Wildcards are allowed in the file specification. If <cDirSpec> is omitted, the default value is *.*.

<cAttributes> specifies inclusion of files with special attributes in the returned information. <cAttributes> is a string containing one or more of the following characters:

DIRECTORY() Attributes

Attribute	Meaning
H	Include hidden files
S	Include system files
D	Include directories
V	Search for the DOS volume label and exclude all other files

Normal files are always included in the search, unless you specify V.

Returns

DIRECTORY() returns an array of subarrays, with each subarray containing information about each file matching <cDirSpec>. The subarray has the following structure:

DIRECTORY() Subarray Structure

Position	Metasymbol	Directry.ch
1	cName	F_NAME
2	cSize	F_SIZE
3	dDate	F_DATE
4	cTime	F_TIME
5	cAttributes	F_ATT

If no files are found matching <cDirSpec> or if <cDirSpec> is an illegal path or file specification, DIRECTORY() returns an empty ({}) array.

Description

DIRECTORY() is an environment function that returns information about files in the current or specified directory. It is similar to ADIR(), but returns a single array instead of adding values to a series of existing arrays passed by reference.

Use DIRECTORY() to perform actions on groups of files. In combination with AEVAL(), you can define a block that can be applied to all files matching the specified <cDirSpec>.

The header file, Directry.ch, in \CLIPPER5\INCLUDE subdirectory contains #defines for the subarray subscripts so references to each file subarray is more readable.

Examples

. This example creates an array of information about files in

```
the current directory and then lists the names of the files using
AEVAL() and QOUT():

#include "Directry.ch"
//
aDirectory := DIRECTORY("*.*", "D")
AEVAL( aDirectory, {|aFile| QOUT(aFile[F_NAME])} )
```

Files: Library is CLIPPER.LIB, header file is Directry.ch.

See Also: AEVAL() CURDIR()

DISKSPACE()
Return the space available on a specified disk

Syntax

DISKSPACE([<nDrive>]) --> nBytes

Arguments

<nDrive> is the number of the drive to query, where one is drive A, two is B, three is C, etc. The default is the current DOS drive if <nDrive> is omitted or specified as zero.

Returns

DISKSPACE() returns the number of bytes of empty space on the specified disk drive as an integer numeric value.

Description

DISKSPACE() is an environment function that determines the number of available bytes remaining on the specified disk drive. It is useful when COPYing or SORTing to another drive to determine if there is enough space available before initiating the operation. You may also use DISKSPACE() with RECSIZE() and RECCOUNT() to create a procedure to back up database files.

DISKSPACE() ignores the SET DEFAULT drive setting.

Examples

. This example is a user-defined function that demonstrates the use of DISKSPACE() to back up a database file to another drive:

```
FUNCTION BackUp( cTargetFile, cTargetDrive )
   LOCAL nSpaceNeeded, nTargetDrive
   //
   nSpaceNeeded := INT((RECSIZE() * ;
      LASTREC()) + HEADER() + 1)
   nTargetDrive := ASC(UPPER(cTargetDrive)) - 64
   //
   IF DISKSPACE(nTargetDrive) < nSpaceNeeded
      RETURN .F.
   ENDIF
   COPY TO (cTargetDrive + ":" + cTargetFile)
   //
   RETURN .T.
```

Files: Library is EXTEND.LIB, source file is EXAMPLEC.C.

See Also: LASTREC() LUPDATE() RECSIZE()

DISPBEGIN()
Begin buffering screen output

Syntax

DISPBEGIN() --> NIL

Returns

DISPBEGIN() always returns NIL.

Description

DISPBEGIN() is a screen function that informs the CA-Clipper display output system that the application is about to perform a series of display operations.

Use DISPBEGIN() with DISPEND() to allow the display output system to buffer display updates. Display output which occurs after DISPBEGIN() but before DISPEND() is allowed to accumulate in internal buffers. When DISPEND() executes, any pending updates appear on the physical display. This is useful in applications where complex screen displays are slow and the appearance of performance is desired.

DISPBEGIN() and DISPEND() calls are optional. They are not required for normal output.

Notes

- **Nested calls:** DISPBEGIN() calls are nested internally. If several DISPBEGIN() calls occur, buffering is allowed until a corresponding number of DISPEND() calls occur.

- **Guaranteed operations:** Display updates performed between DISPBEGIN() and DISPEND() are not guaranteed to be buffered--some updates may become visible before DISPEND() is called. However, all updates are guaranteed to be visible after the closing call to DISPEND().

- **Terminal operations:** Terminal input operations such as INKEY() or READ should not be performed between DISPBEGIN() and DISPEND(). Doing this may cause input or display output to be lost.

- **Incompatible operations:** Display output by other than the CA-Clipper display functions (e.g., by add-on libraries or by DOS via OUTSTD(), etc.) may not be compatible with DISPBEGIN() and DISPEND(). Output may be lost.

Examples

- This example buffers screen output, updates the screen, then displays the buffered screen output:

```
DISPBEGIN()            // Start screen buffering
//
SETPOS(10, 10)
DISPOUT("A display update")
SETPOS(11, 10)
DISPOUT("Another display update")
//
DISPEND()              // Display buffered screen data
```

Files: Library is CLIPPER.LIB.

See Also: DISPEND()

DISPBOX()
Display a box on the screen

Syntax

DISPBOX(<nTop>, <nLeft>, <nBottom>, <nRight>,
[<cnBoxString>], [<cColorString>]) --> NIL

Arguments

<nTop>, **<nLeft>**, **<nBottom>**, and **<nRight>** define the coordinates of the box. DISPBOX() draws a box using row values from zero to MAXROW(), and column values from zero to MAXCOL(). If <nBottom> and <nRight> are larger than MAXROW() and MAXCOL(), the bottom-right corner is drawn off the screen.

<cnBoxString> is a numeric or character expression that defines the border characters of the box. If specified as a numeric expression, a value of 1 displays a single-line box and a value of 2 displays a double-line box. All other numeric values display a single-line box.

If <cnBoxString> is a character expression, it specifies the characters to be used in drawing the box. This is a string of eight border characters and a fill character. If <cnBoxString> is specified as a single character, that character is used to draw the whole box.

If this argument is not specified, a single-line box is drawn.

<cColorString> defines the display color of the box that is drawn. If not specified, the box is drawn using the standard color setting of the current system color as defined by SETCOLOR().

Returns

DISPBOX() always returns NIL.

Description

DISPBOX() is a screen function that draws a box at the specified display coordinates in the specified color. If you specify <cnBoxString>, DISPBOX() draws a box on the screen using configurable border and fill characters. DISPBOX() draws the box using <cnBoxString> starting from the upper lefthand corner, proceeding clockwise and filling the screen region with the ninth character. If the ninth character is not specified, the screen region within the box is not painted. Existing text and color remain unchanged.

After DISPBOX() executes, the cursor is located in the upper corner of the boxed region at <nTop> + 1 and <nLeft> + 1. ROW() and COL() are also updated to reflect the new cursor position.

Note that Box.ch, located in \INCLUDE, provides constants for various border configurations.

Examples

. This code example displays a double-line box using a numeric value to specify the box border:

```
#define B_SINGLE   1
#define B_DOUBLE   2
//
DISPBOX(1, 1, 10, 10, B_DOUBLE, "BG+/B")
```

. This example displays a single-line top and double-line side box by specifying border characters with a manifest constant defined in Box.ch:

```
#include "Box.ch"
//
DISPBOX(1, 1, 10, 10, B_SINGLE_DOUBLE, "BG+/B")
```

Files: Library is CLIPPER.LIB, header file is Box.ch.

See Also: @...BOX @...TO Box characters Colors

DISPCOUNT()
Return the number of pending DISPEND() requests

Syntax

DISPCOUNT() --> nDispCount

Returns

DISPCOUNT() Returns the number of DISPEND() calls required to restore the original display context.

Description

You can use DISPCOUNT() to determine the current display context. CA-Clipper uses display contexts to buffer and and to supervise screen output operations.

Each call to DISPBEGIN() defines a new display context. Output to the display context is suppressed until a matching DISPEND() statement executes.

Since you may nest DISPBEGIN() calls, use DISPCOUNT() to determine whether there are pending screen refresh requests.

Examples

. This example saves the setting of DISPCOUNT(), then releases all pending display contexts before writing to the screen:

```
PROCEDURE ForceDisplay(cExp)
   LOCAL nSavCount

nSavCount := DISPCOUNT()

//  Discard pending display contexts
DO WHILE ( DISPCOUNT() > 0)
   DISPEND()
ENDDO

DISPOUT(cExp)

//  "Rewind" the current display context

DO WHILE (DISPCOUNT() < nSavCount )
   DISPBEGIN()
ENDDO

RETURN
```

Files: Library is CLIPPER.LIB.

See Also: DISPBEGIN() DISPEND()

DISPEND()
Display buffered screen updates

Syntax

DISPEND() --> NIL

Returns

DISPEND() always returns NIL.

Description

DISPEND() is a screen function that informs the CA-Clipper display output system that the application has finished performing a series of display operations.

DISPEND() is used in concert with DISPBEGIN() so the display output system can buffer display updates. This can be important for applications in which complex screen displays are slow and the appearance of performance is desired.

For more information, refer to DISPBEGIN()

Examples

. This example buffers screen output, updates the screen, then displays the buffered screen output:

```
DISPBEGIN()            // Start screen buffering
//
SETPOS(10, 10)
DISPOUT("A display update")
SETPOS(11, 10)
DISPOUT("Another display update")
//
DISPEND()              // Display buffered screen data
```

Files: Library is CLIPPER.LIB.

See Also: DISPBEGIN()

DISPOUT()
Write a value to the display

Syntax

DISPOUT(<exp>, [<cColorString>]) --> NIL

Arguments

<exp> is the value to display.

<cColorString> is an optional argument that defines the display color of <exp>. If unspecified, <exp> is displayed as the standard color of the current system color as defined by SETCOLOR(). <cColorString> is a character expression containing the standard color setting. You can specify a literal color setting, if you enclose it in quote marks.

Returns

DISPOUT() always returns NIL.

Description

DISPOUT() is a simple output function that writes the value of a single expression to the display at the current cursor position. This function ignores the SET DEVICE setting; output always goes to the screen. You can only use this function within a procedure or function.

Examples

. This example performs screen output at a specified location in different colors. Note how the cursor position is saved and restored using ROW(), COL(), and SETPOS():

```
PROCEDURE Showit
   LOCAL nRow, nCol
   ? nCol := COL()           // save original
   ?? nRow := ROW()          // cursor position

   INKEY(2)

   SETPOS(nRow, nCol)
   DISPOUT("This is a test of DISPOUT()")
   ? COL()                   // display current
   ?? ROW()                  // cursor position

   INKEY(2)

   SETPOS(nRow, nCol)
   DISPOUT(space(26))        // clear original position
   SET DEVICE TO PRINTER     // ignores SET DEVICE
   SETPOS(nRow, nCol)        // display at
   DISPOUT("           all through")
                             // original position


   RETURN
```

Files: Library is CLIPPER.LIB.

See Also: COL() OUTSTD() QOUT() ROW() SETPOS() Colors

DOSERROR()
Return the last DOS error number

Syntax

DOSERROR([<nNewOsCode>]) --> nOsCode

Arguments

<nNewOsCode>, if specified, alters the value returned by DOSERROR(). The value must be a numeric value that reflects a DOS error number.

Returns

DOSERROR() returns the DOS error number as an integer numeric value.

Description

DOSERROR() is an error function that returns the last DOS error code associated with an activation of the runtime error block. When a runtime error occurs, the DOSERROR() function is set to the current DOS error if the operation has an associated DOS error. The function value is retained until another runtime error occurs. If the failed operation has no associated DOS error, the DOSERROR() returns zero. With low-level file functions, FERROR() returns the same value as DOSERROR().

Through use of the optional <nNewOsCode>, you may customize to the reporting activation, the returned value for any DOS error.

For a complete list of DOS error numbers and accompanying descriptions, refer to the Error Messages and Appendices guide.

Files: Library is CLIPPER.LIB.

See Also: ERRORBLOCK() FERROR()

DOW()
Convert a date value to a numeric day of the week

Syntax

DOW(<dDate>) --> nDay

Arguments

<dDate> is a date value to convert.

Returns

DOW() returns the day of the week as a number between zero and seven. The first day of the week is one (Sunday) and the last day is seven (Saturday). If <dDate> is empty, DOW() returns zero.

Description

DOW() is a date conversion function that converts a date value to a number identifying the day of the week. It is useful when you want date calculations on a weekly basis. DOW() is similar to CDOW() which returns the day of week as a character string instead of a number.

Examples

. These examples illustrate CDOW() and its relationship to DOW():

```
? DATE()                    // Result: 09/01/89
? DOW(DATE())               // Result: 3
? CDOW(DATE())              // Result: Tuesday
? DOW(DATE() - 2)           // Result: 1
? CDOW(DATE() - 2)          // Result: Sunday
```

. This is a user-defined function that uses DOW() to calculate the date of last Monday from any other date:

```
FUNCTION LastMonday(dDate)
   RETURN (dDate - DOW(dDate) + 2)
```

Files: Library is CLIPPER.LIB.

See Also: CDOW() DATE() DAY()

DTOC()
Convert a date value to a character string

Syntax

DTOC(<dDate>) --> cDate

Arguments

<dDate> is the date value to convert.

Returns

DTOC() returns a character string representation of a date value. The return value is formatted in the current date format. The default format is mm/dd/yy. A null date returns a string of spaces equal in length to the current date format.

Description

DTOC() is a date conversion function used for formatting purposes when you want to display the date in the SET DATE format when a character expression is required (in a LABEL FORM, for example). If you need a specialized date format, you can use TRANSFORM() or a custom expression.

If you are INDEXing a date in combination with a character string, use DTOS() instead of DTOC() to convert the date value to a character string.

Examples

. These examples show general uses of DTOC():

```
? DATE()                        // Result: 09/01/90
? DTOC(DATE())                  // Result: 09/01/90
? "Today is " + DTOC(DATE())
                                // Result: Today is 09/01/90
```

Files: Library is CLIPPER.LIB.

See Also: CTOD() DATE() DTOS() SET CENTURY SET DATE

DTOS()
Convert a date value to a character string formatted as yyyymmdd

---

Syntax

DTOS(<dDate>) --> cDate

Arguments

<dDate> is the date value to convert.

Returns

DTOS() returns a character string eight characters long in the form, yyyymmdd. When <dDate> is a null date (CTOD("")), DTOS() returns a string of eight spaces. The return value is not affected by the current date format.

Description

DTOS() is a date conversion function that is used when creating index expressions consisting of a date value and a character expression. DTOS() converts a date value to a character string that can be concatenated to any other character expression. The return value is structured to preserve date order (year, month, and day).

Examples

. These examples illustrate DTOS() in conjunction with several other functions:

```
? DATE()                      // Result: 09/01/90
? DTOS(DATE())                // Result: 19900901
? LEN(DTOS(CTOD("")))         // Result: 8
```

. This example demonstrates how to create an index with a compound date and character key using DTOS():

```
USE Sales NEW
INDEX ON DTOS(Date) + Salesman TO DateName
```

Files: Library is CLIPPER.LIB.

See Also: CTOD() DATE() DTOC() INDEX

EMPTY()
Determine if the result of an expression is empty

Syntax

EMPTY(<exp>) --> lEmpty

Arguments

<exp> is an expression of any data type.

Returns

EMPTY() returns true (.T.) if the expression results in an empty value; otherwise, it returns false (.F.). The criteria for determining whether a value is considered empty depends on the data type of <exp> according to the following rules:

List of Empty Values

Data Type	Contents
Array	Zero-length
Character	Spaces, tabs, CR/LF, or ("")
Numeric	0
Date	Null(CTOD(""))
Logical	False (.F.)
Memo	Same as character
NIL	NIL

Description

The EMPTY() function has a number of uses. You can use it to determine if a user entered a value into a GET object before committing changes to a database file. It can also determine whether a formal parameter is NIL or unsupplied. In addition, it can test an array for zero-length.

Examples

. These examples illustrate use of EMPTY() against several different data types:

```
? EMPTY(SPACE(5)), EMPTY("")        // Result: .T. .T.
? EMPTY(0), EMPTY(CTOD(""))         // Result: .T. .T.
? EMPTY(.F.), EMPTY(NIL)            // Result: .T. .T.
```

. This example uses EMPTY() to determine whether the user entered a value into the first Get object before writing the new value to the database file:

```
LOCAL cCust := SPACE(15), nAmount := 0
USE Sales NEW
@ 10, 10 GET cCust
@ 11, 10 GET nAmount PICTURE "999.99"
READ
//
IF !EMPTY(cCust)
   APPEND BLANK
   REPLACE Sales->Cust WITH cCust, Sales->Amount ;
      WITH nAmount
ENDIF
```

. This example uses EMPTY() as part of the VALID clause to force the user to enter data into the current Get object:

```
LOCAL cCode := SPACE(5)
```

```
@ 2, 5 SAY "Enter code" GET cCode VALID !EMPTY(cCode)
READ
```

Files: Library is CLIPPER.LIB.

See Also: LEN()

EOF()
Determine when end of file is encountered

Syntax

EOF() --> lBoundary

Returns

EOF() returns true (.T.) when an attempt is made to move the record pointer beyond the last logical record in a database file; otherwise, it returns false (.F.). If there is no database file open in the current work area, EOF() returns false (.F.). If the current database file contains no records, EOF() returns true (.T.).

Description

EOF() is a database function used to test for an end of file boundary condition when the record pointer is moving forward through a database file. Any command that can move the record pointer can set EOF().

The most typical application is as a part of the <lCondition> argument of a DO WHILE construct that sequentially processes records in a database file. Here <lCondition> would include a test for .NOT. EOF(), forcing the DO WHILE loop to terminate when EOF() returns true (.T.).

EOF() and FOUND() are often used interchangeably to test whether a SEEK, FIND, or LOCATE command failed. With these commands, however, FOUND() is preferred.

When EOF() returns true (.T.), the record pointer is positioned at LASTREC() + 1 regardless of whether there is an active SET FILTER or SET DELETED is ON. Further attempts to move the record pointer forward return the same result without error. Once EOF() is set to true (.T.), it retains its value until there is another attempt to move the record pointer.

By default, EOF() operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression (see example below).

Examples

. This example demonstrates EOF() by deliberately moving the record pointer beyond the last record:

```
USE Sales
GO BOTTOM
? EOF()            // Result: .F.
SKIP
? EOF()            // Result: .T.
```

. This example uses aliased expressions to query the value of EOF() in unselected work areas:

```
USE Sales NEW
USE Customer NEW
? Sales->(EOF())
? Customer->(EOF())
```

. This example illustrates how EOF() can be used as part of a condition for sequential database file operations:

```
USE Sales INDEX CustNum NEW
DO WHILE !EOF()
   nOldCust := Sales->CustNum
   nTotalAmount := 0
```

```
      DO WHILE nOldCust = Sales->CustNum .AND. (!EOF())
         ? Sales->CustNum, Sales->Description, ;
               Sales->SaleAmount
         nTotalAmount += Sales->SaleAmount
         SKIP
      ENDDO
      ? "Total amount: ", nTotalAmount
   ENDDO
```

Files: Library is CLIPPER.LIB.

See Also: BOF() FOUND() GO LASTREC() LOCATE SEEK SKIP

ERRORBLOCK()
Post a code block to execute when a runtime error occurs

Syntax

ERRORBLOCK([<bErrorHandler>]) --> bCurrentErrorHandler

Arguments

<bErrorHandler> is the code block to execute whenever a runtime error occurs. When evaluated, the <bErrorHandler> is passed an error object as an argument by the system.

Returns

ERRORBLOCK() returns the current error handling code block. If no error handling block has been posted since the program was invoked, ERRORBLOCK() returns the default error handling block.

Description

ERRORBLOCK() is an error function that defines an error handler to execute whenever a runtime error occurs. Specify the error handler as a code block with the following form:

```
{ |<objError>| <expression list>,... }
```

Where <objError> is an error object containing information about the error. Within the code block, messages can be sent to the error object to obtain information about the error. Returning true (.T.) from the error handling block retries the failed operation and false (.F.) resumes processing.

The error handling code block can be specified either as a list of expressions or as a call to a user-defined function. A call to a user-defined function is more useful since you can use CA-Clipper control statements instead of expressions. This is particularly the case if there is a BEGIN SEQUENCE pending and you want to BREAK to the nearest RECOVER statement.

As this implies, error handling blocks can be used in combination with BEGIN SEQUENCE...END control structures. Within an error handling block, you handle device, low-level, and common errors that have a general recovery mechanism. If the operation needs specific error handling, define a BEGIN SEQUENCE then BREAK to the RECOVER statement, returning the error object for local processing. See the example below.

If no <bErrorHandler> has been specified using ERRORBLOCK() and a runtime error occurs, the default error handling block is evaluated. This error handler displays a descriptive message to the screen, sets the ERRORLEVEL() to 1, then QUITs the program.

Since ERRORBLOCK() returns the current error handling block, it is possible to specify an error handling block for an operation saving the current error handling block, then restore it after the operation has completed. Also, error handlers specified as code blocks, can be passed to procedures and user-defined functions, and RETURNed as values.

For more information on the structure and operations of error objects, refer to the Error class entry in this chapter and the Error Handling Strategies chapter in the Programming and Utilities guide.

Examples

. This code fragment posts, then calls an error handling block when there is an error within a BEGIN SEQUENCE construct:

```
LOCAL bErrorHandler, bLastHandler, objErr
bErrorHandler := { |objError| ;
      MyErrorHandler(objError) }
//
// Save current handler
bLastHandler := ERRORBLOCK(bErrorHandler)
//
BEGIN SEQUENCE
   .
   . <operation statements>
   .
// Receive error object from BREAK
RECOVER USING objErrorInfo
   .
   . <recovery statements>
   .
END
ERRORBLOCK(bLastHandler)   // Restore handler
RETURN

FUNCTION MyErrorHandler( objError )
   //
   BREAK objError          // Return error object to RECOVER
   RETURN NIL
```

Files: Library is CLIPPER.LIB.

See Also: BEGIN SEQUENCE

ERRORLEVEL()
Set the CA-Clipper return code

Syntax

ERRORLEVEL([<nNewReturnCode>]) --> nCurrentReturnCode

Arguments

<nNewReturnCode> is the new return code setting. This can be a value between zero and 255. The default value at startup is zero. If not specified, ERRORLEVEL() reports the current setting without assigning a new value.

Returns

ERRORLEVEL() returns the current CA-Clipper exit code as a numeric value, if one has been set using ERRORLEVEL() with an argument; otherwise, it returns zero.

Description

ERRORLEVEL() is a dual purpose environment function. It returns the current CA-Clipper return code and optionally sets a new return code. The return code is a value set by a child process so the parent process can test the termination state of the child process. Typically, the parent process is DOS and the child process is an application program. Retrieve a return code with the DOS ERRORLEVEL command or INT 21 Function 4Dh.

When a CA-Clipper program terminates, the return code is set to 1 if the process ends with a fatal error. If the process ends normally, the return code is set to zero, or the last ERRORLEVEL() set in the program.

Typically, you would set a return code with ERRORLEVEL() to indicate an error state to the program that invoked the current CA-Clipper program. In most cases this is the application batch file. Here you would test the return code using the DOS ERRORLEVEL command. Refer to your DOS manual for more information.

Notes

. ERRORLEVEL() is not updated after a RUN command terminates. To obtain the return code of the invoked program, you must create an assembler or C routine that queries the child process return code using INT 21 Function 4Dh. Refer to your DOS Technical Reference manual for more information.

Examples

. This example saves the current CA-Clipper return code, then sets a new value:

```
nOldCode := ERRORLEVEL()      // Get current error level
ERRORLEVEL(1)                 // Set new error level
```

. This example uses ERRORLEVEL() to set a return code that can be tested by the parent process:

```
#define ERR_FILE_MISSING      255
#define ERR_POST_INCOMPLETE   254
//
IF !FILE("Sysfile.dbf")
   @ 0, 0
   @ 1, 0
   @ 0, 0 SAY "Fatal error: System ;
               file is missing...quitting"
```

```
      ERRORLEVEL(ERR_FILE_MISSING)
      QUIT
   ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: QUIT

EVAL()
Evaluate a code block

Syntax

EVAL(<bBlock>, [<BlockArg list>]) --> LastBlockValue

Arguments

<bBlock> is the code block to evaluate.

<BlockArg list> is a list of arguments to send to the code block before it is evaluated.

Returns

EVAL() returns the value of the last expression within the block. A code block can return a value of any type.

Description

EVAL() is a code block function. It is the most basic code block evaluation facility in the CA-Clipper system. A code block is a special data value that refers to a piece of compiled program code. For more information on code blocks, refer to the Basic Concepts chapter in the Programming and Utilities guide.

To execute or evaluate a code block, call EVAL() with the block value and any parameters. The parameters are supplied to the block when it is executed. Code blocks may be a series of expressions separated by commas. When a code block is evaluated, the returned value is the value of the last expression in the block.

The CA-Clipper compiler usually compiles a code block at compile time. There are, however, occasions at runtime, when you may need to compile a code block from a character string. You can do this by using the macro operator (&).

EVAL() is often used to create iterator functions. These are functions that apply a block to each member of a data structure. AEVAL(), ASORT(), ASCAN(), and DBEVAL() are iterator functions (e.g., AEVAL() applies a block to each element within an array).

Examples

. This example creates a code block that increments a number then evaluates it:

```
bBlock := { |nArg| nArg + 1 }
? EVAL(bBlock, 1)                    // Result: 2
```

. This example demonstrates compiling a code block at runtime using the macro operator (&):

```
// Compile a string to a block
bBlock := &("{ |nArg| nArg + 1 }")

// Evaluate the block
? EVAL(bBlock, 1)
```

Files: Library is CLIPPER.LIB.

See Also: AEVAL() ASCAN() ASORT() DBEVAL()

EXP()
Calculate e**x

Syntax

EXP(<nExponent>) --> nAntilogarithm

Arguments

<nExponent> is the natural logarithm for which a numeric value is to be calculated.

Returns

EXP() returns a numeric value that is equivalent to the value e raised to the specified power.

Description

EXP() is a mathematical function that calculates the value, y, (the antilogarithm) of the following equation:

ex = y**

Where e is the base of natural logarithms (2.71828...) and x is <nExponent>. The maximum value of <nExponent> is 45 before a numeric overflow occurs. EXP() and LOG() are inverse functions.

The number of decimal places displayed is determined solely by SET DECIMALS regardless of the current SET FIXED value.

Examples

. This example demonstrates several invocations of EXP():

```
? EXP(1)                    // Result: 2.72
SET DECIMALS TO 10
? EXP(1)                    // Result: 2.7182818285
? LOG(EXP(1))               // Result: 1.0000000000
```

Files: Library is CLIPPER.LIB.

See Also: LOG() SET DECIMALS SET FIXED

FCLOSE()
Close an open binary file and write DOS buffers to disk

Syntax

FCLOSE(<nHandle>) --> lError

Arguments

<nHandle> is the file handle obtained previously from FOPEN() or FCREATE().

Returns

FCLOSE() returns false (.F.) if an error occurs while writing; otherwise, it returns true (.T.).

Description

FCLOSE() is a low-level file function that closes binary files and forces the associated DOS buffers to be written to disk. If the operation fails, FCLOSE() returns false (.F.). FERROR() can then be used to determine the reason for the failure. For example, attempting to use FCLOSE() with an invalid handle returns false (.F.), and FERROR() returns DOS error 6, invalid handle. See FERROR() for a complete list of error numbers.

Warning! This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.

Examples

. This example uses FCLOSE() to close a newly created binary file and displays an error message if the close fails:

```
#include "Fileio.ch"
//
nHandle := FCREATE("Testfile", FC_NORMAL)
IF !FCLOSE(nHandle)
   ? "Error closing file, error number: ", FERROR()
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: FCREATE() FERROR() FOPEN()

FCOUNT()
Return the number of fields in the current (.dbf) file

Syntax

FCOUNT() --> nFields

Returns

FCOUNT() returns the number of fields in the database file in the current work area as an integer numeric value. If there is no database file open, FCOUNT() returns zero.

Description

FCOUNT() is a database function. It is useful in applications containing data-independent programs that can operate on any database file. These include generalized import/export and reporting programs. Typically, you use FCOUNT() to establish the upper limit of a FOR...NEXT or DO WHILE loop that processes a single field at a time.

By default, FCOUNT() operates on the currently selected work area.

Examples

. This example illustrates FCOUNT(), returning the number of fields in the current and an unselected work area:

```
USE Sales NEW
USE Customer NEW
? FCOUNT()                    // Result: 5
? Sales->(FCOUNT())           // Result: 8
```

. This example uses FCOUNT() to DECLARE an array with field information:

```
LOCAL aFields := ARRAY(FCOUNT())
AFIELDS(aFields)
```

. This example uses FCOUNT() as the upper boundary of a FOR loop that processes the list of current work area fields:

```
LOCAL nField
USE Sales NEW
FOR nField := 1 TO FCOUNT()
   ? FIELD(nField)
NEXT
```

Files: Library is CLIPPER.LIB.

See Also: FIELDNAME() TYPE()

FCREATE()
Create and/or truncate a binary file to zero-length

Syntax

FCREATE(<cFile>, [<nAttribute>]) --> nHandle

Arguments

<cFile> is the name of the file to create. If the file already exists, its length is truncated to zero without warning.

<nAttribute> is one of the binary file attributes shown in the table below. If this argument is omitted, the default value is zero.

Binary File Attributes

Value	Fileio.ch	Attribute	Description
0	FC_NORMAL	Normal	Create normal read/write file (default)
1	FC_READONLY	Read-only	Create read-only file
2	FC_HIDDEN	Hidden	Create hidden file
4	FC_SYSTEM	System	Create system file

Returns

FCREATE() returns the DOS file handle number of the new binary file in the range of zero to 65,535. If an error occurs, FCREATE() returns -1 and FERROR() is set to indicate an error code.

Description

FCREATE() is a low-level file function that either creates a new file or opens and truncates an existing file. If <cFile> does not exist, it is created and opened for writing. If it does exist and can be opened for writing, it is truncated to zero-length. If it cannot be opened for writing, FCREATE() returns -1 and FERROR() returns the appropriate error value.

When FCREATE() successfully creates a new file, the file is left open in compatibility sharing mode and read/write access mode. The file attribute specified by the <nAttribute> argument is applied to the new file when it is closed, allowing writing to a newly created read-only file. For a list of access modes, see FOPEN().

Since a file handle is required in order to identify an open file to other file functions, always assign the return value from FCREATE() to a variable for later use.

Like other file functions, FCREATE() does not use either the DEFAULT or PATH settings for its operation. Instead, it writes to the current DOS directory unless a path is explicitly stated.

Warning! This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.

Examples

. This example creates a file called Testfile and opens it for reading and writing:

```
#include "Fileio.ch"

//
IF (nHandle := FCREATE("Testfile", FC_NORMAL)) == -1
```

```
      ? "File cannot be created:", FERROR()
      BREAK
   ELSE
      FWRITE(nHandle, "Hello there")
      FCLOSE(nHandle)
   ENDIF
```

Files: Library is CLIPPER.LIB, header file is Fileio.ch.

See Also: FCLOSE() FERROR() FOPEN()

FERASE()
Delete a file from disk

Syntax

FERASE(<cFile>) --> nSuccess

Arguments

<cFile> is the name of the file to be deleted from disk, including extension, optionally preceded by a drive and/or path specification.

Returns

FERASE() returns -1 if the operation fails and zero if it succeeds. In the case of a failure, FERROR() can be used to determine the nature of the error.

Description

FERASE() is a file function that deletes a specified file from disk. FERASE() is the same as the ERASE command but returns a value and can be specified within an expression. When FERASE() is called, <cFile> is deleted from disk only if found in the current DOS directory or in the directory explicitly specified as part of the filename. Like the other file functions and commands, FERASE() does not use either SET DEFAULT or SET PATH to locate <cFile>.

Warning! Files must be CLOSEd before removing them with FERASE().

Examples

. This example deletes a set of files matching a wildcard pattern:

```
#include "Directry.ch"
AEVAL(DIRECTORY("*.BAK"), { |aFile| ;
   FERASE(aFile[F_NAME]) })
```

. This example erases a file and displays a message if the operation fails:

```
IF FERASE("AFile.txt") == -1
   ? "File erase error:", FERROR()
   BREAK
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: CLOSE ERASE FERROR() FRENAME() RENAME

FERROR()
Test for errors after a binary file operation

Syntax

FERROR() --> nErrorCode

Returns

FERROR() returns the DOS error from the last file operation as an integer numeric value. If there is no error, FERROR() returns zero.

FERROR() Return Values

Error	Meaning
0	Successful
2	File not found
3	Path not found
4	Too many files open
5	Access denied
6	Invalid handle
8	Insufficient memory
15	Invalid drive specified
19	Attempted to write to a write-protected disk
21	Drive not ready
23	Data CRC error
29	Write fault
30	Read fault
32	Sharing violation
33	Lock Violation

Description

FERROR() is a low-level file function that indicates a DOS error after a file function is used. These functions include FCLOSE(), FCREATE(), FERASE(), FOPEN(), FREAD(), FREADSTR(), and FRENAME(). FERROR() retains its value until the next execution of a file function.

Warning! This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.

Examples

. This example tests FERROR() after the creation of a binary file and displays an error message if the create fails:

```
#include "Fileio.ch"
//
nHandle := FCREATE("Temp.txt", FC_NORMAL)
IF FERROR() != 0
   ? "Cannot create file, DOS error ", FERROR()
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: DOSERROR() FCLOSE() FCREATE() FOPEN() FWRITE()

FIELDBLOCK()
Return a set-get code block for a given field

Syntax

FIELDBLOCK(<cFieldName>) --> bFieldBlock

Arguments

<cFieldName> is the name of the field to which the set-get block will refer.

Returns

FIELDBLOCK() returns a code block that, when evaluated, sets (assigns) or gets (retrieves) the value of the given field. If <cFieldName> does not exist in the current work area, FIELDBLOCK() returns NIL.

Description

FIELDBLOCK() is a database function that builds a code block. When executed with an argument, the code block created by this function assigns the value of the argument to <cFieldName>. When executed without an argument, the code block retrieves the value of <cFieldName>.

Note that the specified field variable may not exist when the code block is created but must exist before the code block is executed.

Notes

. **Work area:** The code block returned by FIELDBLOCK() sets or gets the value of the specified field in whatever work area is current when the block is run. For example, given work areas 1 and 2, both containing field FName:

```
SELECT 1
FName:= "Kate"
SELECT 2
FName := "Cindy"
bFName := FIELDBLOCK("FName")

SELECT 1
? EVAL(bFName)              // Result: "Kate"
SELECT 2
? EVAL(bFName)              // Result: "Cindy"
```

The function FIELDWBLOCK() provides a set-get block for a field in a specific work area.

Examples

. This example compares FIELDBLOCK() to a code block created using the macro operator. Note that using FIELDBLOCK() avoids the speed and size overhead of the macro operator:

```
// Set-Get block defined using macro operator
bSetGet := &( "{ |setVal| IF( setVal == NIL,;
                FName, FName := setVal ) }" )
// Set-Get block defined using FIELDBLOCK()

// bSetGet created here is the functional
// equivalent of bSetGet above
bSetGet := FIELDBLOCK("FName")
```

Files: Library is CLIPPER.LIB.

See Also: FIELDWBLOCK() MEMVARBLOCK()

FIELDGET()
Retrieve the value of a field using the ordinal position of the field in the database structure

---

Syntax

FIELDGET(<nField>) --> ValueField

Arguments

<nField> is the ordinal position of the field in the record structure for the current work area.

Returns

FIELDGET() returns the value of the specified field. If <nField> does not correspond to the position of any field in the current database file, FIELDGET() returns NIL.

Description

FIELDGET() is a database function that retrieves the value of a field using its position within the database file structure rather than its field name. Within generic database service functions this allows, among other things, the retrieval of field values without use of the macro operator.

Examples

. This example compares FIELDGET() to functionally equivalent code that uses the macro operator to retrieve the value of a field:

```
LOCAL nField := 1, FName, FVal
USE Customer NEW
//
// Using macro operator
FName := FIELD( nField )         // Get field name
FVal := &FName                   // Get field value
// Using FIELDGET()
FVal := FIELDGET( nField )       // Get field value
```

Files: Library is CLIPPER.LIB.

See Also: FIELDPUT()

FIELDNAME()/FIELD()
Return a field name from the current (.dbf) file

Syntax

FIELDNAME/FIELD(<nPosition>) --> cFieldName

Arguments

<nPosition> is the position of a field in the database file structure.

Returns

FIELDNAME() returns the name of the specified field as a character string. If <nPosition> does not correspond to an existing field in the current database file or if no database file is open in the current work area, FIELDNAME() returns a null string ("").

Description

FIELDNAME() is a database function that returns a field name using an index to the position of the field name in the database structure. Use it in data-independent applications where the field name is unknown. If information for more than one field is required, use AFIELDS() to create an array of field information or COPY STRUCTURE EXTENDED to create a database of field information.

If you need additional database file structure information, use TYPE() and LEN(). To obtain the number of decimal places for a numeric field, use the following expression:

```
LEN(SUBSTR(STR(<idField>), RAT(".", ;
              STR(<idField>)) + 1))
```

By default, FIELDNAME() operates on the currently selected work area as shown in the example below.

Examples

. These examples illustrate FIELDNAME() used with several other functions:

```
USE Sales
? FIELDNAME(1)              // Result: BRANCH
? FCOUNT()                  // Result: 5
? LEN(FIELDNAME(0))         // Result: 0
? LEN(FIELDNAME(40))        // Result: 0
```

. This example uses FIELDNAME() to list the name and type of each field in Customer.dbf:

```
USE Customer NEW
FOR nField := 1 TO FCOUNT()
   ? PADR(FIELDNAME(nField), 10),;
          VALTYPE(&(FIELDNAME(nField)))
NEXT
```

. This example accesses fields in unselected work areas using aliased expressions:

```
USE Sales NEW
USE Customer NEW
USE Invoices NEW
//
? Sales->(FIELDNAME(1))            // Result: SALENUM
? Customer->(FIELDNAME(1))         // Result: CUSTNUM
```

Files: Library is CLIPPER.LIB.

See Also: DBSTRUCT() FCOUNT() LEN() VALTYPE()

FIELDPOS()
Return the position of a field in a work area

Syntax

FIELDPOS(<cFieldName>) --> nFieldPos

Arguments

<cFieldName> is the name of a field in the current or specified work area.

Returns

FIELDPOS() returns the position of the specified field within the list of fields associated with the current or specified work area. If the current work area has no field with the specified name, FIELDPOS() returns zero.

Description

FIELDPOS() is a database function that is the inverse of the FIELDNAME() function. FIELDPOS() is most often used with the FIELDPUT() and FIELDGET() functions.

FIELDPOS() return the names of fields in any unselected work area by referring to the function using an aliased expression. See the example below.

Examples

. This example demonstrates a typical specification of the FIELDPOS() function:

```
USE Customer NEW
? FIELDPOS("Name")                    // Result: 1
? FIELDGET(FIELDPOS("Name"))          // Result: Kate
```

. This example uses FIELDPOS() to return the position of a specified field in a unselected work area:

```
USE Customer NEW
USE Invoices NEW
? Customer->(FIELDPOS("Name"))        // Result: 1
? Customer->(FIELDGET(FIELDPOS("Name")))
                                      // Result: Kate
```

Files: Library is CLIPPER.LIB.

See Also: FIELDGET() FIELDPUT()

FIELDPUT()
Set the value of a field variable using the ordinal position of the field in the database structure

---

Syntax

FIELDPUT(<nField>, <expAssign>) --> ValueAssigned

Arguments

<nField> is the ordinal position of the field in the current database file.

<expAssign> is the value to assign to the given field. The data type of this expression must match the data type of the designated field variable.

Returns

FIELDPUT() returns the value assigned to the designated field. If <nField> does not correspond to the position of any field in the current database file, FIELDPUT() returns NIL.

Description

FIELDPUT() is a database function that assigns <expAssign> to the field at ordinal position <nField> in the current work area. This function allows you to set the value of a field using its position within the database file structure rather than its field name. Within generic database service functions this allows, among other things, the setting of field values without use of the macro operator.

Examples

. This example compares FIELDPUT() to functionally equivalent code that uses the macro operator to set the value of a field:

```
// Using macro operator
FName := FIELD(nField)          // Get field name
FIELD->&FName := FVal           // Set field value
// Using FIELDPUT()
FIELDPUT(nField, FVal)          // Set field value
```

Files: Library is CLIPPER.LIB.

See Also: FIELDGET()

FIELDWBLOCK()
Return a set-get code block for a field in a given work area

Syntax

FIELDWBLOCK(<cFieldName>, <nWorkArea>)
--> bFieldWBlock

Arguments

<cFieldName> is the name of the field specified as a character string.

<nWorkArea> is the work area number where the field resides specified as a numeric value.

Returns

FIELDWBLOCK() returns a code block that, when evaluated, sets (assigns) or gets (retrieves) the value of <cFieldName> in the work area designated by <nWorkArea>. If <cFieldName> does not exist in the specified work area, FIELDWBLOCK() returns NIL.

Description

FIELDWBLOCK() is a database function that builds a code block. When evaluated with the EVAL() function, the code block first selects the designated <nWorkArea>. If an argument was passed, the code block then assigns the value of the argument to <cFieldName>. If no argument was passed, the code block retrieves the value of <cFieldName>. The original work area is then reselected before the code block returns control.

Note that the specified field variable may not exist when the code block is created but must exist before the code block is executed.

Notes

. FIELDWBLOCK() is similar to FIELDBLOCK(), except that FIELDBLOCK() incorporates a fixed work area into the set-get block.

Examples

. This example compares FIELDWBLOCK() to a code block created using the macro operator. Note that using FIELDWBLOCK() avoids the speed and size overhead of the macro operator:

```
// Set-Get block for work area 1 defined with
// macro operator
bSetGet := &( "{ |setVal| IF( setVal == NIL, ;
   1->FName, 1->FName := setVal ) }" )
// Set-Get block defined using FIELDWBLOCK()

// bSetGet created here is the functional
// equivalent of bSetGet above
bSetGet := FIELDWBLOCK("FName", 1)
```

Files: Library is CLIPPER.LIB.

See Also: FIELDBLOCK() MEMVARBLOCK()

FILE()
Determine if files exist in the CA-Clipper default directory or path

---

Syntax

FILE(<cFilespec>) --> lExists

Arguments

<cFilespec> is in the current CA-Clipper default directory and path. It is a standard file specification that can include the wildcard characters * and ? as well as a drive and path reference. Explicit references to a file must also include an extension.

Returns

FILE() returns true (.T.) if there is a match for any file matching the <cFilespec> pattern; otherwise, it returns false (.F.).

Description

FILE() is an environment function that determines whether any file matching a file specification pattern is found. FILE() searches the specified directory if a path is explicitly specified.

If a path is not specified, FILE() searches the current CA-Clipper default directory and then the CA-Clipper path. In no case is the DOS path searched. Note also that FILE() does not recognize hidden or system files in its search.

Examples

. In this example FILE() attempts to find Sales.dbf in other than the current CA-Clipper default:

```
? FILE("Sales.dbf")               // Result: .F.
? FILE("\APPS\DBF\Sales.dbf")     // Result: .T.
//
SET PATH TO \APPS\DBF
? FILE("Sales.dbf")               // Result: .T.
//
SET PATH TO
SET DEFAULT TO \APPS\DBF\
? FILE("Sales.dbf")               // Result: .T.
? FILE("*.dbf")                   // Result: .T.
```

Files: Library is CLIPPER.LIB.

See Also: SET DEFAULT SET PATH

FKLABEL()*
Return function key name

Syntax

FKLABEL(<nFunctionKey>) --> cKeyLabel

Returns

FKLABEL() returns a character string representing the name of the function key specified by the numeric argument, <nFunctionKey>. If this argument is less than one or greater than 40, the function returns a null ("") string.

Description

FKLABEL() is a compatibility function used to replicate the FKLABEL() function in dBASE III PLUS. As a general principle, the use of this function is not recommended and not needed in CA-Clipper. The function keys are labeled Fn, where n ranges from one to 40 and corresponds directly to the FKLABEL() argument.

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\FKLABEL.PRG.

See Also: SET FUNCTION

FKMAX()*
Return number of function keys as a constant

Syntax

FKMAX() --> nFunctionKeys

Description

FKMAX() is a compatibility function used to replicate the FKMAX() function in dBASE III PLUS. As a general principle, the use of this function is not recommended and not needed in CA-Clipper. It simply returns a constant value of 40.

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\FKMAX.PRG.

See Also: #define SET FUNCTION

FLOCK()
Lock an open and shared database file

Syntax

FLOCK() --> lSuccess

Returns

FLOCK() returns true (.T.) if an attempt to lock a database file in USE in the current work area succeeds; otherwise, it returns false (.F.). For more information on file locking, refer to the Network Programming chapter in the Programming and Utilities guide.

Description

FLOCK() is a database function used in network environments to lock an open and shared database file, preventing other users from updating the file until the lock is released. Records in the locked file are accessible for read-only operations.

FLOCK() is related to USE...EXCLUSIVE and RLOCK(). USE...EXCLUSIVE opens a database file so that no other user can open the same file at the same time and is the most restrictive locking mechanism in CA-Clipper. RLOCK() is the least restrictive and attempts to place an update lock on a shared record, precluding other users from updating the current record. FLOCK() falls in the middle.

FLOCK() is used for operations that access the entire database file. Typically, these are commands that update the file with a scope or a condition such as DELETE or REPLACE ALL. The following is a list of such commands:

Commands that require an FLOCK()

Command	Mode
APPEND FROM	FLOCK() or USE...EXCLUSIVE
DELETE (multiple records)	FLOCK() or USE...EXCLUSIVE
RECALL (multiple records)	FLOCK() or USE...EXCLUSIVE
REPLACE (multiple records)	FLOCK() or USE...EXCLUSIVE
UPDATE ON	FLOCK() or USE...EXCLUSIVE

For each invocation of FLOCK(), there is one attempt to lock the database file, and the result is returned as a logical value. A file lock fails if another user currently has a file or record lock for the same database file or EXCLUSIVE USE of the database file. If FLOCK() is successful, the file lock remains in place until you UNLOCK, CLOSE the DATABASE, or RLOCK().

By default, FLOCK() operates on the currently selected work area as shown in the example below.

Notes

- **SET RELATION:** CA-Clipper does not automatically lock all work areas in the relation chain when you lock the current work area, and an UNLOCK has no effect on related work areas.

Examples

- This example uses FLOCK() for a batch update of prices in Inventory.dbf:

```
USE Inventory NEW
IF FLOCK()
```

```
   REPLACE ALL Inventory->Price WITH ;
         Inventory->Price * 1.1
ELSE
   ? "File not available"
ENDIF
```

. This example uses an aliased expression to attempt a file lock in an unselected work area:

```
USE Sales NEW
USE Customer NEW
//
IF !Sales->(FLOCK())
   ? "Sales is in use by another"
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: RLOCK() SET EXCLUSIVE* UNLOCK USE

FOPEN()
Open a binary file

Syntax

FOPEN(<cFile>, [<nMode>]) --> nHandle

Arguments

<cFile> is the name of the file to open including the path if there is one.

<nMode> is the requested DOS open mode indicating how the opened file is to be accessed. The open mode is composed of elements from the two types of modes described in the tables below. If just the Access Mode is used, the file is opened non-sharable. The default open mode is zero, which indicates non-sharable and read-only.

FOPEN() Access Modes

Mode	Fileio.ch	Operation
0	FO_READ	Open for reading (default)
1	FO_WRITE	Open for writing
2	FO_READWRITE	Open for reading or writing

The Sharing Modes determine how other processes may access the file.

The Access Modes in combination (+) with the Sharing modes determine the accessibility of the file in a network environment.

FOPEN() Sharing Modes

Mode	Fileio.ch	Operation
0	FO_COMPAT	Compatibility mode (default)
16	FO_EXCLUSIVE	Exclusive use
32	FO_DENYWRITE	Prevent others from writing
48	FO_DENYREAD	Prevent others from reading
64	FO_DENYNONE	Allow others to read or write
64	FO_SHARED	Same as FO_DENYNONE

Returns

FOPEN() returns the file handle of the opened file in the range of zero to 65,535. If an error occurs, FOPEN() returns -1.

Description

FOPEN() is a low-level file function that opens an existing binary file for reading and writing, depending on the <nMode> argument. Whenever there is an open error, FERROR()to return the DOS error number. For example, if the file does not exist, FOPEN() returns -1 and FERROR() returns 2 to indicate that the file was not found. See FERROR() for a complete list of error numbers.

If the specified file is opened successfully, the value returned is the DOS handle for the file. This value is similar to an alias in the database system and is required to identify the open file to other file functions. It is, therefore, important to assign the return value to a variable for later use as in the example below.

Warning! This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.

Notes

- **Accessing files in other directories:** FOPEN() does not obey either SET DEFAULT or SET PATH. Instead, it searches the current DOS directory and path setting unless a path is explicitly stated as part of the <cFile> argument.

Examples

- This example uses FOPEN() to open a file with sharable read/write status and displays an error message if the open fails:

```
#include "Fileio.ch"
//
nHandle := FOPEN("Temp.txt", FO_READWRITE + FO_SHARED)
IF FERROR() != 0
   ? "Cannot open file, DOS error ", FERROR()
   BREAK
ENDIF
```

Files: Library is CLIPPER.LIB, header file is Fileio.ch.

See Also: FCLOSE() FCREATE() FERROR()

AADD()
Add a new element to the end of an array

Syntax

AADD(<aTarget>, <expValue>) --> Value

Arguments

<aTarget> is the array to add a new element to.

<expValue> is the value assigned to the new element.

Returns

AADD() evaluates <expValue> and returns its value. If <expValue> is not specified, AADD() returns NIL.

Description

AADD() is an array function that increases the actual length of the target array by one. The newly created array element is assigned the value specified by <expValue>.

AADD() is used to dynamically grow an array. It is useful for building dynamic lists or queues. A good example of this is the GetList array used by the GET system to hold Get objects. After a READ or CLEAR GETS, GetList becomes an empty array. Each time you execute an @...GET command, the GET system uses AADD() to add a new element to the end of the GetList array, and then assigns a new Get object to the new element.

AADD() is similar to ASIZE() but only adds one element at a time; ASIZE() can grow or shrink an array to a specified size. AADD(), however, has the advantage that it can assign a value to the new element, while ASIZE() cannot. AADD() may also seem similar to AINS(), but they are different: AINS() moves elements within an array, but it does not change the array's length.

Note: If <expValue> is another array, the new element in the target array will contain a reference to the array specified by <expValue>.

Examples

. These examples demonstrate the effects of multiple invocations of AADD() to an array:

```
aArray := {}               // Result: aArray is an empty array
AADD(aArray, 5)            // Result: aArray is { 5 }
AADD(aArray, 10)           // Result: aArray is { 5, 10 }
AADD(aArray, { 12, 10 })   // Result: aArray is
                           // { 5, 10, { 12, 10 } }
```

Files: Library is CLIPPER.LIB.

See Also: AINS() ASIZE()

FOUND()
Determine if the previous search operation succeeded

Syntax

FOUND() --> lSuccess

Returns

FOUND() returns true (.T.) if the last search command was successful; otherwise, it returns false (.F.).

Description

FOUND() is a database function that determines whether a search operation (i.e., FIND, LOCATE, CONTINUE, SEEK, or SET RELATION) succeeded. When any of these commands are executed, FOUND() is set to true (.T.) if there is a match; otherwise, it is set to false (.F.).

If the search command is LOCATE or CONTINUE, a match is the next record meeting the scope and condition. If the search command is FIND, SEEK or SET RELATION, a match is the first key in the controlling index that equals the search argument. If the key value equals the search argument, FOUND() is true (.T.); otherwise, it is false (.F.).

The value of FOUND() is retained until another record movement command is executed. Unless the command is another search command, FOUND() is automatically set to false (.F.).

Each work area has a FOUND() value. This means that if one work area has a RELATION set to a child work area, querying FOUND() in the child returns true (.T.) if there is a match.

By default, FOUND() operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression (see example below).

FOUND() will return false (.F.) if there is no database open in the current work area.

Examples

. This example illustrates the behavior of FOUND() after a record movement command:

```
USE Sales INDEX Sales
? INDEXKEY(0)            // Result: SALESMAN
SEEK "1000"
? FOUND()                // Result: .F.
SEEK "100"
? FOUND()                // Result: .T.
SKIP
? FOUND()                // Result: .F.
```

. This example tests a FOUND() value in an unselected work area using an aliased expression:

```
USE Sales INDEX Sales NEW
USE Customer INDEX Customer NEW
SET RELATION TO CustNum INTO Sales
//
SEEK "Smith"
? FOUND(), Sales->(FOUND())
```

. This code fragment processes all Customer records with the key value "Smith" using FOUND() to determine when the key value changes:

```
USE Customer INDEX Customer NEW
SEEK "Smith"
DO WHILE FOUND()
   .
   . <statements>
   .
   SKIP
   LOCATE REST WHILE Name == "Smith"
ENDDO
```

Files: Library is CLIPPER.LIB.

See Also: EOF() LOCATE SEEK SET RELATION SET SOFTSEEK

FREAD()
Read characters from a binary file into a buffer variable

Syntax

FREAD(<nHandle>, @<cBufferVar>, <nBytes>) --> nBytes

Arguments

<nHandle> is the file handle obtained from FOPEN(), FCREATE(), or predefined by DOS.

<cBufferVar> is the name of an existing and initialized character variable used to store data read from the specified file. The length of this variable must be greater than or equal to <nBytes>. <cBufferVar> must be passed by reference and, therefore, must be prefaced by the pass-by-reference operator (@).

<nBytes> is the number of bytes to read into the buffer.

Returns

FREAD() returns the number of bytes successfully read as an integer numeric value. A return value less than <nBytes> or zero indicates end of file or some other read error.

Description

FREAD() is a low-level file function that reads characters from a binary file into an existing character variable. It reads from the file starting at the current DOS file pointer position, advancing the file pointer by the number of bytes read. All characters are read including control, null, and high-order (above CHR(127)) characters.

FREAD() is similar in some respects to both FREADSTR() and FSEEK(). FREADSTR() reads a specified number of bytes from a file up to the next null (CHR(0)) character. FSEEK() moves the file pointer without reading.

If there is an error during the file read, FERROR() returns the DOS error number. See FERROR() for the list of error numbers.

Warning! This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.

Examples

. This example uses FREAD() after successfully opening a file to read 128 bytes into a buffer area:

```
#define F_BLOCK      128

//
cBuffer := SPACE(F_BLOCK)
nHandle := FOPEN("Temp.txt")
//
IF FERROR() != 0
   ? "File open error:", FERROR()
ELSE
   IF FREAD(nHandle, @cBuffer, F_BLOCK) <> F_BLOCK
      ? "Error reading Temp.txt"
   ENDIF
   FCLOSE(nHandle)
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: BIN2I() BIN2L() BIN2W() FERROR() FWRITE()

FREADSTR()
Read characters from a binary file

Syntax

FREADSTR(<nHandle>, <nBytes>) --> cString

Arguments

<nHandle> is the file handle obtained from FOPEN(), FCREATE(), or predefined by DOS.

<nBytes> is the number of bytes to read, beginning at the current DOS file pointer position.

Returns

FREADSTR() returns a character string up to 65,535 (64K) bytes. A null return value ("") indicates an error or end of file.

Description

FREADSTR() is a low-level file function that reads characters from an open binary file beginning with the current DOS file pointer position. Characters are read up to <nBytes> or until a null character (CHR(0)) is encountered. All characters are read including control characters except for CHR(0). The file pointer is then moved forward <nBytes>. If <nBytes> is greater than the number of bytes from the pointer position to the end of the file, the file pointer is positioned to the last byte in the file.

Warning! This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.

Examples

. This example displays the ASCII values of the first 16 bytes of a text file:

```
#include "Fileio.ch"
//
nHandle := FOPEN("New.txt", FC_NORMAL)
IF FERROR() != 0
   ? "File open error:", FERROR()
ELSE
   cString := FREADSTR(nHandle, 16)
   ? cString
   FCLOSE(nHandle)
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: BIN2I() BIN2L() BIN2W() FERROR() FREAD() FSEEK()

FRENAME()
Change the name of a file

Syntax

FRENAME(<cOldFile>, <cNewFile>) --> nSuccess

Arguments

<cOldFile> is the name of the file to rename, including the file extension. A drive letter and/or path name may also be included as part of the filename.

<cNewFile> is the new name of the file, including the file extension. A drive letter and/or path name may also be included as part of the name.

Returns

FRENAME() returns -1 if the operation fails and zero if it succeeds. In the case of a failure, FERROR() can be used to determine the nature of the error.

Description

FRENAME() is a file function that changes the name of a specified file to a new name and is identical to the RENAME command.

When FRENAME() is called, <cOldFile> is renamed only if it is located in the current DOS directory or in the specified path. FRENAME() does not use SET DEFAULT or SET PATH to locate <cOldFile>.

If the source directory is different from the target directory, the file moves to the target directory. In the instance that either <cNewFile> exists or is currently open, FRENAME() fails and returns -1, indicating that it did not perform its designated action. The nature of the error can be determined with FERROR().

Warning! Files must be CLOSEd before renaming. Attempting to rename an open file will produce unpredictable results. When a database file is renamed, the associated memo (.dbt) file must also be renamed. Failure to do so may compromise the integrity of your databases.

Examples

. This example demonstrates a file rename:

```
IF FRENAME("OldFile.txt", "NewFile.txt") == -1
   ? "File error:", FERROR()
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: CLOSE ERASE FERASE() FERROR() FILE() RENAME

FSEEK()
Set a binary file pointer to a new position

Syntax

FSEEK(<nHandle>, <nOffset>, [<nOrigin>]) --> nPosition

Arguments

<nHandle> is the file handle obtained from FOPEN(), FCREATE(), or predefined by DOS.

<nOffset> is the number of bytes to move the file pointer from the position defined by <nOrigin>. It can be a positive or negative number. A positive number moves the pointer forward, and a negative number moves the pointer backward in the file.

<nOrigin> defines the starting location of the file pointer before FSEEK() is executed. The default value is zero, representing the beginning of file. If <nOrigin> is the end of file, <nOffset> must be zero or negative.

Methods of Moving the File Pointer

Origin	Fileio.ch	Description
0	FS_SET	Seek from beginning of file
1	FS_RELATIVE	Seek from the current pointer position
2	FS_END	Seek from end of file

Returns

FSEEK() returns the new position of the file pointer relative to the beginning of file (position 0) as an integer numeric value. This value is without regard to the original position of the file pointer.

Description

FSEEK() is a low-level file function that moves the file pointer forward or backward in an open binary file without actually reading the contents of the specified file. The beginning position and offset are specified as function arguments, and the new file position is returned. Regardless of the function arguments specified, the file pointer cannot be moved beyond the beginning or end of file boundaries.

Warning! This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system.

Examples

. This example uses FSEEK() to determine the length of a file by seeking from the end of file. Then, the file pointer is reset to the beginning of file:

```
#include "Fileio.ch"
//
// Open the file read-only
IF (nHandle := FOPEN("Temp.txt")) >= 0
   //
   // Get length of the file
   nLength := FSEEK(nHandle, 0, FS_END)
   //
   // Reset file position to beginning of file
   FSEEK(nHandle, 0)
   FCLOSE(nHandle)
```

```
ELSE
   ? "File open error:", FERROR()
ENDIF
```

- This pseudofunction positions the file pointer at the last byte in a binary file:

```
#define FileBottom(nHandle);
      (FSEEK(nHandle, 0, FS_END))
```

- This pseudofunction positions the file pointer at the first byte in a binary file:

```
#define FileTop(nHandle);
      (FSEEK(nHandle, 0))
```

- This pseudofunction reports the current position of the file pointer in a specified binary file:

```
#define FilePos(nHandle);
      (FSEEK(nHandle, 0, FS_RELATIVE))
```

Files: Library is CLIPPER.LIB, header file is Fileio.ch.

See Also: FCREATE() FERROR() FOPEN() FREAD() FREADSTR() FWRITE()

FWRITE()
Write to an open binary file

Syntax

FWRITE(<nHandle>, <cBuffer>, [<nBytes>])
--> nBytesWritten

Arguments

<nHandle> is the file handle obtained from FOPEN(), FCREATE(), or predefined by DOS.

<cBuffer> is the character string to write to the specified file.

<nBytes> indicates the number of bytes to write beginning at the current file pointer position. If omitted, the entire content of <cBuffer> is written.

Returns

FWRITE() returns the number of bytes written as an integer numeric value. If the value returned is equal to <nBytes>, the operation was successful. If the return value is less than <nBytes> or zero, either the disk is full or another error has occurred.

Description

FWRITE() is a low-level file function that writes data to an open binary file from a character string buffer. You can either write all or a portion of the buffer contents. Writing begins at the current file position, and the function returns the actual number of bytes written.

If FWRITE() results in an error condition, FERROR() can be used to determine the specific error.

Warning! This function allows low-level access to DOS files and devices. It should be used with extreme care and requires a thorough knowledge of the operating system

Examples

. This example copies the contents of one file to another:

```
#include "Fileio.ch"
#define F_BLOCK   512
//
cBuffer := SPACE(F_BLOCK)
nInfile := FOPEN("Temp.txt", FO_READ)
nOutfile := FCREATE("Newfile.txt", FC_NORMAL)
lDone := .F.
//
DO WHILE !lDone

   nBytesRead := FREAD(nInfile, @cBuffer, F_BLOCK)
   IF FWRITE(nOutfile, cBuffer, nBytesRead) < ;
               nBytesRead
      ? "Write fault: ", FERROR()
      lDone := .T.
   ELSE
      lDone := (nBytesRead == 0)
   ENDIF
ENDDO
//
FCLOSE(nInfile)
FCLOSE(nOutfile)
```

Files: Library is CLIPPER.LIB.

See Also: FCLOSE() FCREATE() FERROR() FOPEN() I2BIN() L2BIN()

GETENV()
Retrieve the contents of a DOS environment variable

Syntax

GETENV(<cEnvironmentVariable>) --> cString

Arguments

<cEnvironmentVariable> is the name of the DOS environment variable. When specifying this argument, you can use any combination of upper and lowercase letters; GETENV() is not case-sensitive.

Returns

GETENV() returns the contents of the specified DOS environment variable as a character string. If the variable cannot be found, GETENV() returns a null string ("").

Description

GETENV() is an environment function that lets you retrieve information from the DOS environment into an application program. Typically, this is configuration information, including path names, that give the location of files (database, index, label, or reports). This function is particularly useful for network environments.

Notes

- **Empty return value:** If you are certain that an environment variable exists and yet GETENV() always returns a null string (""), be sure there are no spaces between the environment variable name and the first character of the string assigned to it in the DOS SET command.

- **Compatibility:** In previous releases of CA-Clipper, the function was called GETE(). This abbreviated form of GETENV() is still operational.

Examples

- This example retrieves the current DOS PATH setting, making it the current CA-Clipper PATH:

```
cPath := GETENV("PATH")
SET PATH TO (cPath)
```

- This example uses environment variables to configure the specific locations of files. When you set up a system, define environment variables that contain the location of various file types as well as the CLIPPER environment variable (see The Runtime Environment chapter in the Programming and Utilities guide), like this:

```
C>SET LOC_DBF=<database file path>
C>SET LOC_NTX=<index file path>
C>SET LOC_RPT=<report file path>
```

  In the configuration section of your application program, assign the contents of the environment variables to variables. Then when you access a file, preface the reference with the path variable as follows:

```
cDdfDirectory := GETENV("LOC_DBF")
USE (cDdfDirectory + "Invoices")
```

Files: Library is CLIPPER.LIB.

HARDCR()
Replace all soft carriage returns in a character string with hard carriage returns

---

Syntax

HARDCR(<cString>) --> cConvertedString

Arguments

<cString> is the character string or memo field to convert.

Returns

HARDCR() returns a character string up to 65,535 (64K) characters in length.

Description

HARDCR() is a memo function that replaces all soft carriage returns (CHR(141)) with hard carriage returns (CHR(13)). It is used to display long character strings and memo fields containing soft carriage returns with console commands. In CA-Clipper, console commands (including REPORT and LABEL FORM) do not automatically convert soft carriage returns to hard carriage returns, making it necessary for you to explicitly make the conversion. Soft carriage returns are added by MEMOEDIT() when lines wrap.

Notes

. **HARDCR() in REPORT and LABEL FORMs:** If HARDCR() is used in a REPORT FORM or LABEL FORM contents expression and not used elsewhere, you must declare it EXTERNAL to assure that it is linked.

Examples

. To display a memo field formatted with the automatic word wrapping of MEMOEDIT():

```
USE Sales NEW
? HARDCR(Sales->Notes)
```

Files: Library is EXTEND.LIB.

See Also: ?|?? EXTERNAL* LABEL FORM MEMOTRAN() REPORT FORM STRTRAN()

HEADER()
Return the current database file header length

Syntax

HEADER() --> nBytes

Returns

HEADER() returns the number of bytes in the header of the current database file as an integer numeric value. If no database file is in use, HEADER() returns a zero (0).

Description

HEADER() is a database function that is used with LASTREC(), RECSIZE(), and DISKSPACE() to create procedures for backing up files.

By default, HEADER() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

Examples

. This example determines the header size of the Sales.dbf:

```
USE Sales NEW
? HEADER()            // Result: 258
```

. This example defines a pseudofunction, DbfSize(), that uses HEADER() with RECSIZE() and LASTREC() to calculate the size of the current database file in bytes:

```
#define DbfSize()   ((RECSIZE() * LASTREC()) + ;
      HEADER() + 1)
```

Later you can use DbfSize() as you would any function:

```
USE Sales NEW
USE Customer NEW
? DbfSize()
? Sales->(DbfSize())
```

Files: Library is EXTEND.LIB.

See Also: DISKSPACE() LASTREC() RECSIZE()

I2BIN()
Convert a CA-Clipper numeric to a 16-bit binary integer

Syntax

I2BIN(<nInteger>) --> cBinaryInteger

Arguments

<nInteger> is an integer numeric value to convert. Decimal digits are truncated.

Returns

I2BIN() returns a two-byte character string containing a 16-bit binary integer.

Description

I2BIN() is a low-level file function that converts an integer numeric value to a character string formatted as a binary integer--least significant byte first. I2BIN() is used with FWRITE() to convert a CA-Clipper numeric to a standard binary form. The inverse of I2BIN() is BIN2I().

Examples

. This example opens a database file using low-level file functions and writes a new date of the last update to bytes 1-3:

```
#include "Fileio.ch"
//
nHandle = FOPEN("Sales.dbf", FO_READWRITE)
//
// Convert date of last update to int
nYear = I2BIN(90)
nMonth = I2BIN(12)
nDay = I2BIN(15)
//
// Point to the date of last update
FSEEK(nHandle, 1, FS_SET)
//
// Write the new update date using only the first byte
FWRITE(nHandle, nYear, 1)
FWRITE(nHandle, nMonth, 1)
FWRITE(nHandle, nDay, 1)
FCLOSE(nHandle)
```

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM.

See Also: BIN2I() BIN2L() BIN2W() CHR() FWRITE() L2BIN()

IF()
Return the result of an expression based on a condition

Syntax

[I]IF(<lCondition>, <expTrue>, <expFalse>) --> Value

Arguments

<lCondition> is a logical expression to be evaluated.

<expTrue> is the value, a condition-expression, of any data type, returned if <lCondition> is true (.T.).

<expFalse> is the value, of any date type, returned if <lCondition> is false (.F.). This argument need not be the same data type as <expTrue>.

Returns

IF() returns the evaluation of <expTrue> if <lCondition> evaluates to true (.T.) and <expFalse> if it evaluates to false (.F.). The value returned is the data type of the valid condition-expression.

Description

IF() is a logical conversion function. It is one of the most powerful and versatile functions in CA-Clipper. It provides a mechanism to evaluate a condition within an expression. With this ability you can convert a logical expression to another data type.

Examples

. This example converts a logical data value to a numeric data value:

```
lPaid = .T.
? IF(lPaid, 1, 0)               // Result: 1
```

. In this example a logical field is formatted depending on whether the Customer is past due or not:

```
@ ROW() + 1, 25 SAY IF(lPaid, SPACE(10), "Go get 'em")
```

. If you are printing forms, you can print an indicating symbol in different columns depending on the value of a logical field:

```
@ ROW(), IF(InHospital, 10, 12) SAY "X"
```

. You can also use IF() to force the LABEL FORM to print blank lines. Enter the following expression when you create the label with RL.EXE:

```
IF(EMPTY(Company), CHR(255), Company)
```

Files: Library is CLIPPER.LIB.

See Also: DO CASE IF

INDEXEXT()
Return the default index extension based on the database driver currently linked

---

Syntax

INDEXEXT() --> cExtension

Returns

Unless you have linked another database driver, INDEXEXT() returns ".NTX" to indicate that the default CA-Clipper driver is in effect. If the dBASE III PLUS compatible database driver is linked, the function returns ".NDX".

Description

INDEXEXT() returns the default index file extension by determining which database driver is currently linked.

Examples

. In this example, INDEXEXT() creates an existence test for the Customer index file independent of the database driver linked into the current program:

```
USE Customer NEW
//
IF .NOT. FILE("Customer" + INDEXEXT())
   INDEX ON CustName TO Customer
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: INDEXKEY() INDEXORD()

INDEXKEY()
Return the key expression of a specified index

Syntax

INDEXKEY(<nOrder>) --> cKeyExp

Arguments

<nOrder> is the ordinal position of the index in the list of index files opened by the last USE...INDEX or SET INDEX TO command for the current work area. A zero value specifies the controlling index, without regard to its actual position in the list.

Returns

INDEXKEY() returns the key expression of the specified index as a character string. If there is no corresponding index or if no database file is open, INDEXKEY() returns a null string ("").

Description

INDEXKEY() is a database function that determines the key expression of a specified index in the current work area and returns it as a character string. To evaluate the key expression, specify INDEXKEY() as a macro expression like this: &(INDEXKEY(<nOrder>)).

INDEXKEY() has a number of applications, but two specific instances are important. Using INDEXKEY(), you can TOTAL on the key expression of the controlling index without having to specify the key expression in the source code. The other instance occurs within a DBEDIT() user function. Here, you may want to determine whether or not to update the screen after the user has edited a record. Generally, it is only necessary to update the screen if the key expression of the controlling index has changed for the current record. Both of these examples are illustrated below.

By default, INDEXKEY() operates on the currently selected work area. It can be made to operate on an unselected work area by specifying it within an aliased expression (see example below).

Examples

. This example accesses the key expression of open indexes in the current work area:

```
#define ORD_NATURAL   0
#define ORD_NAME      1
#define ORD_SERIAL    2
//
USE Customer INDEX Name, Serial NEW
SET ORDER TO ORD_SERIAL
? INDEXKEY(ORD_NAME)        // Result: Name index exp
? INDEXKEY(ORD_SERIAL)      // Result: Serial index exp
? INDEXKEY(ORD_NATURAL)     // Result: Serial index exp
```

. This example accesses the key expression of the controlling index in an unselected work area:

```
USE Customer INDEX Name, Serial NEW
USE Sales INDEX Salesman NEW
? INDEXKEY(0), Customer->(INDEXKEY(0))
```

. This example uses INDEXKEY() as part of a TOTAL ON key expression. Notice that INDEXKEY() is specified using a macro expression to force evaluation of the expression:

```
USE Sales INDEX Salesman NEW
TOTAL ON &(INDEXKEY(0)) FIELDS SaleAmount TO ;
      SalesSummary
```

. This example uses INDEXKEY() to determine whether the DBEDIT() screen should be updated after the user has edited the current field value. Generally, you must update the DBEDIT() screen if the user changes a field that is part of the controlling index key. FieldEdit() is a user-defined function called from a DBEDIT() user function to edit the current field if the user has pressed an edit key:

```
#include "Dbedit.ch"
#define ORD_NATURAL   0
FUNCTION FieldEdit()
   LOCAL indexVal
   // Save current key expression and value
   indexVal = &(INDEXKEY(ORD_NATURAL))
   .
   . <code to GET current field value>
   .
   // Refresh screen if key value has changed
   IF indexVal != &(INDEXKEY(ORD_NATURAL))
      nRequest = DE_REFRESH
   ELSE
      nRequest = DE_CONT
   ENDIF
   RETURN nRequest
```

Files: Library is CLIPPER.LIB.

See Also: INDEX INDEXEXT() INDEXORD() SET INDEX SET ORDER USE

INDEXORD()
Return the order position of the controlling index

Syntax

INDEXORD() --> nOrder

Returns

INDEXORD() returns an integer numeric value. The value returned is equal to the position of the controlling index in the list of open indexes for the current work area. A value of zero indicates that there is no controlling index and records are being accessed in natural order. If no database file is open, INDEXORD() will also return a zero.

Description

INDEXORD() is a database function that determines the position of the controlling index in the list of index files opened by the last USE...INDEX or SET INDEX TO in the current work area. It is often useful to save the last controlling index so it can be restored later.

By default, INDEXORD() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

Examples

. This example uses INDEXORD() to save the current order. After changing to a new order, it uses the saved value to restore the original order:

```
USE Customer INDEX Name, Serial NEW
nOrder := INDEXORD()                  // Result: 1
SET ORDER TO 2
? INDEXORD()                          // Result: 2
SET ORDER TO nOrder
? INDEXORD()                          // Result: 1
```

. This example uses an aliased expression to determine the order number of the controlling index in an unselected work area:

```
USE Sales INDEX Salesman, CustNum NEW
USE Customer INDEX Name, Serial NEW
? Sales->(INDEXORD())                 // Result: 1
```

Files: Library is CLIPPER.LIB.

See Also: INDEX INDEXKEY() SET INDEX SET ORDER USE

INKEY()
Extract a character from the keyboard buffer

Syntax

INKEY([<nSeconds>]) --> nInkeyCode

Arguments

<nSeconds> specifies the number of seconds INKEY() waits for a keypress. You can specify the value in increments as small as one-tenth of a second. Specifying zero halts the program until a key is pressed. If <nSeconds> is omitted, INKEY() does not wait for a keypress.

Returns

INKEY() returns an integer numeric value from -39 to 386, identifying the key extracted from the keyboard buffer. If the keyboard buffer is empty, INKEY() returns zero. INKEY() returns values for all ASCII characters, function, Alt-function, Ctrl-function, Alt-letter, and Ctrl-letter key combinations.

Description

INKEY() is a keyboard function that extracts the next key pending in the keyboard buffer and returns a value representing that key. The value is also saved internally and can be accessed using LASTKEY(). If the <nSeconds> argument is specified and there are no pending keys in the buffer, program execution pauses until a key appears in the keyboard buffer, or <nSeconds> has elapsed. The time INKEY() waits is based on the operating system clock and is not related to the microprocessor speed. If <nSeconds> is zero, program execution pauses until a key is placed into the buffer. Note that INKEY() is not a wait state and therefore SET KEYs are not active.

INKEY() is similar to the NEXTKEY() function. Unlike, INKEY(), however, NEXTKEY() reads but does not extract the key from the keyboard buffer. This is useful when you need to test for a key without processing it.

INKEY() is the basic primitive of the CA-Clipper system for fetching keys from the keyboard buffer. It is used for polling the keyboard or pausing program execution to wait for a keypress. As an example, you can use INKEY() to terminate commands with a record scope such as LIST, LABEL FORM, and REPORT FORM by including it in a WHILE condition. Refer to the example below.

For a complete list of INKEY() codes and Inkey.ch constants, refer to the Error Messages and Appendices guide.

Examples

. This example gets a key from the keyboard then displays its character value followed by the INKEY() value:

```
#include "Inkey.ch"
//
LOCAL nInkeyCode := 0
DO WHILE LASTKEY() != K_ESC
   ? "Press any key: "
   nInkeyCode := INKEY(0)
   ?? "Character:", CHR(nInkeyCode),;
      "INKEY() code:", LTRIM(STR(nInkeyCode))
ENDDO
RETURN
```

. This example uses INKEY() to poll for a user interrupt keypress during a REPORT FORM. If the user presses Esc during the

printing process, the REPORT FORM terminates:

```
#include "Inkey.ch"
//
USE Sales INDEX Salesman NEW
REPORT FORM Monthly FOR MONTH(SalesDate) = ;
      MONTH(DATE());
   WHILE INKEY() != K_ESC
```

Files: Library is CLIPPER.LIB, header file is Inkey.ch.

See Also: CHR() LASTKEY() NEXTKEY() SET KEY Inkey codes

ABS()
Return the absolute value of a numeric expression

Syntax

ABS(<nExp>) --> nPositive

Arguments

<nExp> is the numeric expression to evaluate.

Returns

ABS() returns a number representing the absolute value of its argument. The return value is a positive number or zero.

Description

ABS() is a numeric function that determines the magnitude of a numeric value independent of its sign. It lets you, for example, obtain the difference between two numbers as a positive value without knowing in advance which of the two is larger.

As a formalism, ABS(x) is defined in terms of its argument, x, as follows: if x >= 0, ABS(x) returns x; otherwise, ABS(x) returns the negation of x.

Examples

. These examples show typical results from ABS():

```
nNum1 := 100
nNum2 := 150
? nNum1 - nNum2                // Result: -50
? ABS(nNum1 - nNum2)           // Result: 50
? ABS(nNum2 - nNum1)           // Result: 50
? ABS(-12)                     // Result: 12
? ABS(0)                       // Result: 0
```

Files: Library is CLIPPER.LIB.

INT()
Convert a numeric value to an integer

Syntax

INT(<nExp>) --> nInteger

Arguments

<nExp> is a numeric expression to convert to an integer.

Returns

INT() returns an integer numeric value.

Description

INT() is a numeric function that converts a numeric value to an integer by truncating--not rounding--all digits to the right of the decimal point. INT() is useful in operations where the decimal portion of a number is not needed.

Examples

. These examples demonstrate the results of various invocations of the INT() function:

```
? INT(100.00)              // Result: 100
? INT(.5)                  // Result: 0
? INT(-100.75)             // Result: -100
```

Files: Library is CLIPPER.LIB.

See Also: ROUND()

ISALPHA()
Determine if leftmost character in a string is alphabetic

Syntax

ISALPHA(<cString>) --> lBoolean

Arguments

<cString> is the character string to examine.

Returns

ISALPHA() returns true (.T.) if the first character in <cString> is alphabetic; otherwise, it returns false (.F.).

Description

ISALPHA() is a character function that determines if the specified string begins with an alphabetic character. An alphabetic character consists of any uppercase or lowercase letter from A to Z. ISALPHA() returns false (.F.) if the string begins with a digit or any other character.

Examples

. These examples demonstrate various results of ISALPHA():

```
? ISALPHA("AbcDe")              // Result: .T.
? ISALPHA("aBcDE")              // Result: .T.
? ISALPHA("1BCde")              // Result: .F.
? ISALPHA(".FRED")              // Result: .F.
```

Files: Library is EXTEND.LIB.

See Also: ISDIGIT() ISLOWER() ISUPPER() LOWER() UPPER()

ISCOLOR()
Determine if the current computer has color capability

Syntax

ISCOLOR() | ISCOLOUR() --> lBoolean

Returns

ISCOLOR() returns true (.T.) if there is a color graphics card installed; otherwise, it returns false (.F.).

Description

ISCOLOR() is a screen function that allows you to make decisions about the type of screen attributes to assign (color or monochrome). Note that some monochrome adapters with graphics capability return true (.T.).

Examples

. This example installs color attribute variables at runtime:

```
IF ISCOLOR()
   cBox  = "BG+/B, W/N"
   cSays = "BG/B, W/N"
   cGets = "W/N, N/W"
ELSE
   cBox  = "W+"
   cSays = "W/N, N+/W"
   cGets = "W/N, N/W"
ENDIF
.
. <statements>
.
SETCOLOR(cSays)
```

Files: Library is CLIPPER.LIB.

See Also: SETCOLOR()

ISDIGIT()
Determine if leftmost character in a character string is a digit

---

Syntax

ISDIGIT(<cString>) --> lBoolean

Arguments

<cString> is the character string to examine.

Returns

ISDIGIT() returns true (.T.) if the first character of the character string is a digit between zero and nine; otherwise, it returns false (.F.).

Description

ISDIGIT() is a character function that determines whether the first character in a string is a numeric digit between zero and nine. If any other character is the first character of the <cString>, ISDIGIT() returns false (.F.).

ISDIGIT() is useful where you need to know if the current character string is a number before converting it to a numeric value with the VAL() function.

Examples

. These examples demonstrate various results of ISDIGIT():

```
? ISDIGIT("AbcDe")          // Result: .F.
? ISDIGIT("1abcd")          // Result: .T.
? ISDIGIT(".12345")         // Result: .F.
```

Files: Library is EXTEND.LIB.

See Also: ISALPHA() ISLOWER() ISUPPER() LOWER() UPPER()

ISLOWER()
Determine if leftmost character is a lowercase letter

Syntax

ISLOWER(<cString>) --> lBoolean

Arguments

<cString> is the character string to examine.

Returns

ISLOWER() returns true (.T.) if the first character of the character string is a lowercase letter; otherwise, it returns false (.F.).

Description

ISLOWER() is a character function that determines whether the first character of a character string is lowercase. It is the inverse of ISUPPER() which determines whether a character begins with an uppercase character.

Both ISLOWER() and ISUPPER() relate to the LOWER() and UPPER() functions which actually convert lowercase characters to uppercase, and vice versa.

Examples

. These examples demonstrate various results of ISLOWER():

```
? ISLOWER("aBcDe")         // Result: .T.
? ISLOWER("AbcDe")         // Result: .F.
? ISLOWER("1abcd")         // Result: .F.
? ISLOWER("abcd")          // Result: .T.
```

Files: Library is CLIPPER.LIB.

See Also: ISALPHA() ISDIGIT() ISUPPER() LOWER() UPPER()

ISPRINTER()
Determine whether the LPT1 is ready

Syntax

ISPRINTER() --> lReady

Returns

ISPRINTER() returns true (.T.) if LPT1 is ready; otherwise, it returns false (.F.).

Description

ISPRINTER() is a printer function that determines whether the parallel port (LPT1) is online and ready to print. ISPRINTER() is hardware-dependent and, therefore, only works on IBM BIOS compatible systems.

You can check ISPRINTER() to make sure the printer is ready before you begin a print operation; however, if an error occurs during the print operation, a runtime error is generated.

Examples

. This example tests the parallel port for readiness with up to 25 retries. If the parallel port is ready, the printer operation begins:

```
LOCAL nCount := 0, nTimes := 25, lReady
//
DO WHILE nCount++ <= nTimes .AND. !(lReady := ;
      ISPRINTER())
ENDDO
//
IF lReady
   REPORT FORM Sales TO PRINTER
ELSE
   ? "Printer not ready..."
   BREAK
ENDIF
```

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM.

See Also: SET DEVICE SET PRINTER

ISUPPER()
Determine if leftmost character in a string is uppercase

Syntax

ISUPPER(<cString>) --> lBoolean

Arguments

<cString> is the character string to examine.

Returns

ISUPPER() returns true (.T.) if the first character is an uppercase letter; otherwise, it returns false (.F.).

Description

ISUPPER() is a character function that determines whether the first character of a string is uppercase. It is the inverse of ISLOWER(). Both ISUPPER() and ISLOWER() relate to the UPPER() and LOWER() functions which actually convert uppercase characters to lowercase, and vice versa.

Examples

. These examples illustrate ISUPPER() applied to various values:

```
? ISUPPER("AbCdE")          // Result: .T.
? ISUPPER("aBCdE")          // Result: .F.
? ISUPPER("$abcd")          // Result: .F.
? ISUPPER("8ABCD")          // Result: .F.
```

Files: Library is CLIPPER.LIB.

See Also: ISALPHA() ISDIGIT() ISLOWER() LOWER() UPPER()

L2BIN()
Convert a CA-Clipper numeric value to a 32-bit binary integer

---

Syntax

L2BIN(<nExp>) --> cBinaryInteger

Arguments

<nExp> is the numeric value to convert. Decimal digits are truncated.

Returns

L2BIN() returns a four-byte character string formatted as a 32-bit binary integer.

Description

L2BIN() is a low-level file function used with FWRITE() to write CA-Clipper numeric values to a binary file. This function is like I2BIN() which formats a CA-Clipper numeric to a 16-bit binary value.

L2BIN() is the inverse function of BIN2L().

Examples

. This example creates a new binary file then writes a series of numbers to the files using L2BIN() to convert the numeric value to 32-bit binary form:

```
#include "Fileio.ch"
//
LOCAL nNumber, nHandle
nHandle := FCREATE("MyFile", FC_NORMAL)
FOR nNumber := 1 TO 100
   FWRITE(nHandle, L2BIN(nNumber) + CHR(0))
NEXT
FCLOSE(nHandle)
```

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM.

See Also: BIN2I() BIN2L() BIN2W() CHR() FWRITE() I2BIN()

LASTKEY()
Return the INKEY() value of the last key extracted from the keyboard buffer

---

Syntax

LASTKEY() --> nInkeyCode

Returns

LASTKEY() returns a number between -39 and 386, identifying the INKEY() value of the last key extracted from the keyboard buffer.

Description

LASTKEY() is a keyboard function that reports the INKEY() value of the last key fetched from the keyboard buffer by the INKEY() function, or a wait state such as ACCEPT, INPUT, READ, WAIT, ACHOICE(), DBEDIT(), or MEMOEDIT(). LASTKEY() retains its current value until another key is fetched from the keyboard buffer.

LASTKEY() has a number of uses which include:

. Determining the key that terminates a READ

. Determining the key that exits the current Get object within a user-defined function, invoked by a VALID clause

. Identifying an exception key in the user function of ACHOICE(), DBEDIT(), or MEMOEDIT()

LASTKEY() is also used with UPDATED() to determine if any Get object's buffer was changed during a READ.

LASTKEY() is related to NEXTKEY() and READKEY(). NEXTKEY() reads the current key pending in the keyboard buffer without removing it. Use NEXTKEY() instead of INKEY() when polling for a key.

For a complete list of INKEY() codes and Inkey.ch constants for each key, refer to the Error Messages and Appendices guide.

Examples

. This example illustrates a typical application of LASTKEY() to test the key that exits a READ. If the user exits with any key other than Esc and a GET was changed, the specified database file is updated:

```
#include "Inkey.ch"
//
USE Customer NEW
MEMVAR->balance = Customer->Balance
@ 10, 10 SAY "Current Balance" GET MEMVAR->balance
READ
//
IF (LASTKEY() != K_ESC) .AND. UPDATED()
   REPLACE Customer->Balance WITH MEMVAR->balance
ENDIF
```

Files: Library is CLIPPER.LIB, header file is Inkey.ch.

See Also: CHR() INKEY() KEYBOARD NEXTKEY() Inkey codes

LASTREC()
Determine the number of records in the current (.dbf) file

Syntax

LASTREC() | RECCOUNT()* --> nRecords

Returns

LASTREC() returns the number of physical records in the current database file as an integer numeric value. Filtering commands such as SET FILTER or SET DELETED have no effect on the return value. LASTREC() returns zero if there is no database file in USE in the current work area.

Description

LASTREC() is a database function that determines the number of physical records in the current database file. LASTREC() is identical to RECCOUNT() which is supplied as a compatibility function.

By default, LASTREC() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

Examples

. This example illustrates the relationship between LASTREC(), RECCOUNT(), and COUNT:

```
USE Sales NEW
? LASTREC(), RECCOUNT()          // Result: 84 84
//
SET FILTER TO Salesman = "1001"
COUNT TO nRecords
? nRecords, LASTREC()            // Result: 14 84
```

. This example uses an aliased expression to access the number of records in a open database file in an unselected work area:

```
USE Sales NEW
USE Customer NEW
? LASTREC(), Sales->(LASTREC())
```

Files: Library is CLIPPER.LIB.

See Also: COUNT EOF()

LEFT()
Extract a substring beginning with the first character in a string

---

Syntax

LEFT(<cString>, <nCount>) --> cSubString

Arguments

<cString> is a character string from which to extract characters. The maximum size of <cString> is 65,535 (64K) bytes.

<nCount> is the number of characters to extract.

Returns

LEFT() returns the leftmost <nCount> characters of <cString> as a character string. If <nCount> is negative or zero, LEFT() returns a null string (""). If <nCount> is larger than the length of the character string, LEFT() returns the entire string.

Description

LEFT() is a character function that returns a substring of a specified character string. It is the same as SUBSTR(<cString>, 1, <nCount>). LEFT() is also like RIGHT() which returns a substring beginning with the last character in a string.

LEFT(), RIGHT(), and SUBSTR() are often used with both the AT() and RAT() functions to locate the first and/or the last position of a substring before extracting it.

Examples

. This example extracts the first three characters from the left of the target string:

```
? LEFT("ABCDEF", 3)                 // Result: ABC
```

. This example extracts a substring from the beginning of a string up to the first occurrence of a comma:

```
LOCAL cName := "James, William"
? LEFT(cName, AT(",", cName) - 1)   // Result: James
```

Files: Library is CLIPPER.LIB.

See Also: AT() LTRIM() RAT() RIGHT() RTRIM() STUFF() SUBSTR()

LEN()
Return the length of a character string or the number of elements in an array

---

Syntax

LEN(<cString> | <aTarget>) --> nCount

Arguments

<cString> is the character string to count.

<aTarget> is the array to count.

Returns

LEN() returns the length of a character string or the number of elements in an array as an integer numeric value. If the character string is a null string ("") or the array is empty, LEN() returns zero.

Description

LEN() is a character and array function that returns the length of a character string or the number of elements in an array. With a character string, each byte counts as one, including an embedded null byte (CHR(0)). By contrast, a null string (""), counts as zero.

For an array, LEN() returns the number of elements. If the array is multidimensional, subarrays count as one element. This means that the LEN() of a nested or multidimensional array simply returns the length of the first dimension. To determine the number of elements in other dimensions, use LEN() on the subarrays as shown in the example below. Note that nested arrays in CA-Clipper need not have uniform dimensions.

Examples

. These examples demonstrate LEN() with various arguments:

```
? LEN("string of characters")        // Result: 20
? LEN("")                            // Result: 0
? LEN(CHR(0))                        // Result: 1
//
LOCAL aTest[10]
? LEN(aTest)                         // Result: 10
```

. This example creates a literal two-dimensional array then returns the number of elements in the subarray contained in the first element of the original array:

```
LOCAL aArray := { {1, 2}, {1, 2}, {1, 2} }
? LEN(aArray)                        // Result: 3
? LEN(aArray[1])                     // Result: 2
```

. This example navigates a multidimensional array using LEN():

```
LOCAL aArray := { {1, 2}, {1, 2}, {1, 2} }
LOCAL nRow, nColumn, nRowCount, nColumnCount
//
nRowCount = LEN(aArray)
FOR nRow = 1 TO nRowCount
   nColumnCount = LEN(aArray[nRow])
   FOR nColumn = 1 TO nColumnCount
      ? nRow, nColumn, aArray[nRow][nColumn]
   NEXT
NEXT
```

. In this example a function returns an array of numeric values that describe the dimensions of a nested or multidimensional array.

The function assumes that the array has uniform dimensions:

```
FUNCTION Dimensions( aArray )

   LOCAL aDims := {}
   DO WHILE ( VALTYPE(aArray) == "A" )
      AADD( aDims, LEN(aArray) )
      aArray := aArray[1]
   ENDDO
   RETURN (aDims)
```

Files: Library is CLIPPER.LIB.

See Also: AADD() ASIZE() LTRIM() RTRIM()

ACHOICE()
Execute a pop-up menu

Syntax

```
ACHOICE(<nTop>, <nLeft>, <nBottom>, <nRight>,
   <acMenuItems>,
   [<alSelectableItems> | <lSelectableItems>],
   [<cUserFunction>],
   [<nInitialItem>],
   [<nWindowRow>]) --> nPosition
```

Arguments

<nTop>, **<nLeft>** and **<nBottom>**, **<nRight>** are the upper left and lower right window coordinates. Row values can range from zero to MAXROW() and column values can range from zero to MAXCOL().

<acMenuItems> is an array of character strings to display as the menu items. The individual menu items are later identified by their numeric positions in this array.

<alSelectableItems> is a parallel array of logical values--one element for each item in <acMenuItems>--that specify the selectable menu items. Elements can be logical values or character strings. ACHOICE() will not permit a null string and stops displaying if it encounters one. If the element is a character string, it is evaluated as a macro expression which should evaluate to a logical data type. In either case, a value of false (.F.) means that the corresponding menu item is not available, and a value of true (.T.) means that it is available. If you specify <lSelectableItems> instead of an array, false (.F.) makes all menu items unavailable and true (.T.) makes all menu items available. By default, all menu items are available for selection.

<cUserFunction> is the name of a user-defined function that executes when an unrecognizable key is pressed. Specify the function name as a character expression without parentheses or arguments. Note that the behavior of ACHOICE() is affected by the presence of this argument. Refer to the discussion below for further information.

<nInitialItem> is the position in the <acMenuItems> array of the item that will be highlighted when the menu is initially displayed. If you specify an unavailable menu item or no argument at all, the initial menu item is the first selectable item in the array.

<nWindowRow> is the window row number on which the initial menu item will appear. Row numbering begins with zero. By default, the initial menu item appears as close to the top of the window as possible, without leaving any empty rows at the bottom. Thus, if there are enough menu items following the initial one to fill up the window, the initial form will appear on the first row (row zero) of the menu. This function argument is used to control the initial menu appearance when there are more menu items than will fit in the window.

As with all functions, optional arguments are omitted by using a comma instead of the actual argument.

Returns

ACHOICE() returns the numeric position in the <acMenuItems> array of the menu item selected. If the selection process is aborted, ACHOICE() returns zero.

Description

ACHOICE() is a user interface function that can create various kinds of pop-up menus. Each menu uses an array of character strings as menu

items and a parallel array of logical values to determine whether items are selectable. When you invoke ACHOICE(), the list of menu items is displayed within the specified window coordinates. When the user presses Return, the current item is selected, and ACHOICE() returns the position of the menu item in <acMenuItems>. When the user presses Esc, ACHOICE() aborts and returns zero.

The menu items scroll if the number of items in <acMenuItems> exceeds the number of rows in the menu window, and the user attempts to move the highlight beyond the top or bottom of the menu window. Note that the highlight does not wrap when you reach the top or bottom of the list of items. Pressing the first letter does, however, wrap the highlight within the set of items whose first letter matches the key you press.

. **Navigating the menu:** ACHOICE() has two modes depending on whether the <cUserFunction> argument is specified. If it is not specified the following navigation keys are active:

ACHOICE() Keys (No User Function)

Key	Action
Uparrow	Go to previous item
Dnarrow	Go to next item
Home	Go to first item in menu
End	Go to last item in menu
Ctrl-Home	Go to first item in window
Ctrl-End	Go to last item in window
PgUp	Go to previous page
PgDn	Go to next page
Ctrl-PgUp	Go to the first item in menu
Ctrl-PgDn	Go to last item in menu
Return	Select current item
Esc	Abort selection
Leftarrow	Abort selection
Rightarrow	Abort selection
First Letter	Go to next item beginning with first letter

. **Color:** Menu items are displayed in standard color, the highlight in enhanced color, and the unavailable items in the unselected color. For example, the following color statement:

```
SETCOLOR("W+/N, BG+/B, , , W/N")
```

displays a menu that is bright white on black, the highlight is bright cyan on blue, and the unavailable menu items are dim white on black.

. **User function:** Like the other user interface functions, ACHOICE() supports a user function. The user function is specified when you want to nest ACHOICE() invocations to create hierarchical menus or to redefine keys.

When a user function is specified, ACHOICE() processes only a limited set of keys automatically. These are listed in the table below. All other keys generate a key exception which passes control to the user function for handling. Control is also passed to the user function when ACHOICE() goes idle (i.e., when there are no more keys to process).

ACHOICE() Keys (User Function Specified)

Key	Action
Uparrow	Go to previous item
Dnarrow	Go to next item
Ctrl-Home	Go to first item in window

```
Ctrl-End       Go to last item in window
PgUp           Go to previous page
PgDn           Go to next page
Ctrl-PgUp      Go to the first item in menu
Ctrl-PgDn      Go to last item in menu
```

When ACHOICE() executes the user function, it automatically passes the following three parameters:

- The current ACHOICE() mode
- The current element in the array of items
- The relative row position within the menu window

The mode indicates the current state of ACHOICE() depending on the key pressed and the action taken by ACHOICE() prior to executing the user function. The mode parameter has the following possible values:

ACHOICE() Modes

Mode	Achoice.ch	Description
0	AC_IDLE	Idle
1	AC_HITTOP	Attempt to cursor past top of list
2	AC_HITBOTTOM	Attempt to cursor past bottom of list
3	AC_EXCEPT	Keystroke exceptions
4	AC_NOITEM	No selectable items

After the user function has performed whatever operations are appropriate to the ACHOICE() mode or LASTKEY(), it must RETURN a value requesting ACHOICE() to perform an operation from the following set of actions:

ACHOICE() User Function Return Values

Value	Achoice.ch	Action
0	AC_ABORT	Abort selection
1	AC_SELECT	Make selection
2	AC_CONT	Continue ACHOICE()
3	AC_GOTO	Go to the next item whose first character matches the key pressed

Examples

. This example uses two literal arrays to specify the menu items and selection criteria. After the menu is displayed and the user makes a selection, the name of the selected menu item is displayed:

```
acMenuItems := {"One", "Two", "-------", "Three"}
alSelectableItems := {.T., .T., .F., .T.}
nPosition := ACHOICE(10, 10, 12, 15, acMenuItems,;
                     alSelectableItems)
? acMenuItems[nPosition]
```

. This example declares an array of menu items and supplies a user-defined function which displays a message with each highlighted choice:

```
#include "achoice.ch"
#include "inkey.ch"

PROCEDURE Main()
```

```
   LOCAL acMenuItems[4], cUserFunction, nRetVal
   LOCAL nKey, nPos

   acMenuItems[1] := "Add"
   acMenuItems[2] := "Edit"
   acMenuItems[3] := "Delete"
   acMenuItems[4] := "Update"

   CLS

   nPos := ACHOICE( 10, 10, 13, 15, acMenuItems,;
                   .T., "cUserFunction" )
   DO CASE
   CASE nPos == 1
      //  Put ADD routine here
   CASE nPos == 2
      //  Put EDIT routine here
   CASE nPos == 3
      //  Put DELETE routine here
   CASE nPos ==4
      //  Put UPDATE routine here
   ENDCASE

RETURN


FUNCTION cUserFunction( nMode, nCurElement, nRowPos )

   LOCAL nRetVal := AC_CONT     // Default, Continue
   LOCAL nKey := LASTKEY()

   DO CASE
// After all pending keys are processed, display message
   CASE nMode == AC_IDLE
   DO CASE
      CASE nCurElement == 1
         @ 22, 5 SAY " Adding   "
      CASE nCurElement == 2
         @ 22, 5 SAY " Editing  "
      CASE nCurElement ==  3
         @ 22, 5 SAY " Deleting "
      CASE nCurElement ==  4
         @ 22, 5 SAY " Updating "
   ENDCASE

      nRetVal := AC_CONT            // Continue ACHOICE()

   CASE nMode == AC_HITTOP          // Attempt to go past Top
      TONE( 100, 3 )
   CASE nMode == AC_HITBOTTOM       // Attempt to go past
                                    // Bottom
      TONE( 100, 3 )

   CASE nMode == AC_EXCEPT          // Key Exception
      DO CASE
      CASE nKey == K_RETURN         // If RETURN key, select
         nRetVal := AC_SELECT
      CASE nKey == K_ESC            // If ESCAPE key, abort
         nRetVal := AC_ABORT
      OTHERWISE
            nRetVal := AC_GOTO      // Otherwise, go to item
      ENDCASE
   ENDCASE

RETURN nRetVal
```

. The next example declares the arrays, specifies a selection condition for one of the menu items, and supplies a user function:

```
EXTERNAL UPDATED
//
FUNCTION MyMenu
   LOCAL acMenuItems[4], alSelectableItems[4],;
          cUserFunction := "DoIt"
   //
   acMenuItems[1] := "Add Record"
   acMenuItems[2] := "Edit Record"
   acMenuItems[3] := "Delete Record"
   acMenuItems[4] := "Update Record"
   //
   alSelectableItems[1] := .T.
   alSelectableItems[2] := .T.
   alSelectableItems[3] := .T.
   alSelectableItems[4] := "!UPDATED()"
   // Selection condition

   RETURN ACHOICE(10, 10, 12, 15, acMenuItems,;
      alSelectableItems, cUserFunction)
```

. This example uses two arrays to specify menu items and corresponding action blocks. After the menu is displayed and the user makes a selection, the ACHOICE() return value is used to determine which action block of the aActionItems array is evaluated:

```
PROCEDURE Main()
   LOCAL nChoice
   LOCAL aMenuItems := { "Add Record   ", ;
                         "Edit Record  ", ;
                         "Delete Record", ;
                         "Update Record"   }

   LOCAL aActionItems := { {|| AddFunc()  }, ;
                           {|| EditFunc() }, ;
                           {|| DelFunc()  }, ;
                           {|| UpdFunc()  }  }

   nChoice := ACHOICE( 10, 10, 13, 22, aMenuItems )

   IF nChoice == 0
      QUIT      // ESCAPE was pressed
   ENDIF

   EVAL( aActionItems[nChoice] )

RETURN
```

Files: Library is EXTEND.LIB, header files are Achoice.ch and Inkey.ch.

See Also: MENU TO SETCOLOR() Inkey codes

LOG()
Calculate the natural logarithm of a numeric value

Syntax

LOG(<nExp>) --> nNaturalLog

Arguments

<nExp> is a numeric value greater than zero to convert to its natural logarithm.

Returns

LOG() returns the natural logarithm as a numeric value. If <nExp> is less than or equal to zero, LOG() returns a numeric overflow (displayed as a row of asterisks).

Description

LOG() is a numeric function that calculates the natural logarithm of a number and is the inverse of EXP(). The natural logarithm has a base of e which is approximately 2.7183. The LOG() function returns x in the following equation:

```
e**x = y
```

where y is the numeric expression used as the LOG() argument (i.e., LOG(y) = x). Due to mathematical rounding, the values returned by LOG() and EXP() may not agree exactly (i.e., EXP(LOG(x)) may not always equal x).

Examples

. These examples demonstrate various results of LOG():

```
? LOG(10)                // Result: 2.30
? LOG(10 * 2)            // Result: 3.00
? EXP(LOG(1))            // Result: 1.00
? LOG(2.71)              // Result: 1.00
```

. This example is a user-defined function that returns the base 10 logarithm:

```
FUNCTION Log10( nNumber )
IF nNumber > 0
   RETURN LOG(nNumber)/LOG(10)
ELSE
   RETURN NIL
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: EXP() SET DECIMALS SET FIXED

LOWER()
Convert uppercase characters to lowercase

Syntax

LOWER(<cString>) --> cLowerString

Arguments

<cString> is a character string to convert to lowercase.

Returns

LOWER() returns a copy of <cString> with all alphabetic characters converted to lowercase. All other characters remain the same as in the original string.

Description

LOWER() is a character function that converts upper and mixedcase strings to lowercase. It is related to UPPER() which converts lower and mixedcase strings to uppercase. LOWER() is related to the ISLOWER() and ISUPPER() functions which determine whether a string begins with a lower or uppercase letter.

LOWER() is generally used to format character strings for display purposes. It can, however, be used to normalize strings for case-independent comparison or INDEXing purposes.

Examples

. These examples demonstrate various results of LOWER():

```
? LOWER("STRING")             // Result: string
? LOWER("1234 CHARS = ")      // Result: 1234 chars =
```

Files: Library is CLIPPER.LIB.

See Also: ISLOWER() ISUPPER() UPPER()

LTRIM()
Remove leading spaces from a character string

Syntax

LTRIM(<cString>) --> cTrimString

Arguments

<cString> is the character string to copy without leading spaces.

Returns

LTRIM() returns a copy of <cString> with the leading spaces removed. If <cString> is a null string ("") or all spaces, LTRIM() returns a null string ("").

Description

LTRIM() is a character function that formats character strings with leading spaces. These can be, for example, numbers converted to character strings using STR().

LTRIM() is related to RTRIM() which removes trailing spaces, and ALLTRIM() which removes both leading and trailing spaces. The inverse of ALLTRIM(), LTRIM(), and RTRIM() are the PADC(), PADR(), and PADL() functions which center, right-justify, or left-justify character strings by padding them with fill characters.

Examples

. These examples illustrate LTRIM() used with several other functions:

```
nNumber = 18
? STR(nNumber)                // Result:         18
? LEN(STR(nNumber))           // Result: 10
? LTRIM(STR(nNumber))         // Result: 18
? LEN(LTRIM(STR(nNumber)))    // Result: 2
```

Files: Library is CLIPPER.LIB.

See Also: ALLTRIM() PAD() RTRIM() STR() SUBSTR()

LUPDATE()
Return the last modification date of a (.dbf) file

Syntax

LUPDATE() --> dModification

Returns

LUPDATE() returns the date of last change to the open database file in the current work area. If there is no database file in USE, LUPDATE() returns a blank date.

Description

LUPDATE() is a database function that determines the date the database file in the current work area was last modified and CLOSEd. By default, LUPDATE() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression as shown in the example below.

Examples

. This example demonstrates that the modification date of database file is not changed until the database file is closed:

```
? DATE()                  // Result: 09/01/90
USE Sales NEW
? LUPDATE()               // Result: 08/31/90
//
APPEND BLANK
? LUPDATE()               // Result: 08/31/90
CLOSE DATABASES
//
USE Sales NEW
? LUPDATE()               // Result: 09/01/90
```

. This example uses an aliased expression to access LUPDATE() for a database file open in an unselected work area:

```
USE Sales NEW
USE Customer NEW
? LUPDATE(), Sales->(LUPDATE())
```

Files: Library is EXTEND.LIB.

See Also: FIELDNAME() LASTREC() RECSIZE()

MAX()
Return the larger of two numeric or date values

Syntax

MAX(<nExp1>, <nExp2>) --> nLarger
MAX(<dExp1>, <dExp2>) --> dLarger

Arguments

<nExp1> and **<nExp2>** are the numeric values to compare.

<dExp1> and **<dExp2>** are the date values to compare.

Returns

MAX() returns the larger of the two arguments. The value returned is the same type as the arguments.

Description

MAX() is a numeric and a date function that ensures the value of an expression is larger than a specified minimum. The inverse of MAX() is MIN(), which returns the lesser of two numeric or date values.

Examples

. In these examples MAX() returns the greater of two numeric values:

```
? MAX(1, 2)                    // Result: 2
? MAX(2, 1)                    // Result: 2
```

. In these examples MAX() compares date values:

```
? DATE()                       // Result: 09/01/90
? MAX(DATE(), DATE() + 30)     // Result: 10/01/90
? MAX(DATE(), CTOD(""))        // Result: 09/01/90
```

Files: Library is CLIPPER.LIB.

See Also: MIN()

MAXCOL()
Determine the maximum visible screen column

Syntax

MAXCOL() --> nColumn

Returns

MAXCOL() returns the column number of the rightmost visible column for display purposes.

Description

MAXCOL() is a screen function that determines the maximum visible column of the screen. Row and column numbers start at zero in CA-Clipper.

If you use a C or other extended function to set the video mode, use the SETMODE() function so your CA-Clipper application returns the correct value for MAXCOL().

Examples

. This example uses MAXROW() and MAXCOL() to determine the area to draw a box then executes DBEDIT() within the box region:

```
CLS
@ 0, 0 TO MAXROW(), MAXCOL() DOUBLE
DBEDIT(1, 1, MAXROW() + 1, MAXCOL() - 1)
```

Files: Library is CLIPPER.LIB.

See Also: COL() MAXROW() ROW()

MAXROW()
Determine the maximum visible screen row

Syntax

MAXROW() --> nRow

Returns

MAXROW() returns the row number of the bottommost visible row for display purposes.

Description

MAXROW() is a screen function that determines the maximum visible row of the screen. Row and column numbers start at zero in CA-Clipper.

If you use a C or other extended function to set the video mode, use the SETMODE() function so your CA-Clipper application returns the correct value for MAXCOL().

Examples

. This user-defined function, ScreenSize(), uses MAXROW() and MAXCOL() to return an array containing the current screen size:

```
FUNCTION ScreenSize
   RETURN { MAXROW(), MAXCOL() }
```

Files: Library is CLIPPER.LIB.

See Also: COL() MAXCOL() ROW()

MEMOEDIT()
Display or edit character strings and memo fields

Syntax

```
MEMOEDIT([<cString>],
   [<nTop>], [<nLeft>],
   [<nBottom>], [<nRight>],
   [<lEditMode>],
   [<cUserFunction>],
   [<nLineLength>],
   [<nTabSize>],
   [<nTextBufferRow>],
   [<nTextBufferColumn>],
   [<nWindowRow>],
   [<nWindowColumn>]) --> cTextBuffer
```

Arguments

<cString> is the character string or memo field to copy to the MEMOEDIT() text buffer. If not specified, the text buffer is empty.

<nTop>, **<nLeft>**, **<nBottom>**, and **<nRight>** are the upper left and lower right window coordinates. Row values can range from zero to MAXROW(), and column positions can range from zero to MAXCOL(). If not specified, the default coordinates are 0, 0, MAXROW(), and MAXCOL().

<lEditMode> determines whether the text buffer can be edited or merely displayed. Specifying true (.T.) allows the user to make changes to the text buffer, while specifying false (.F.) only allows the user to browse the text buffer. If not specified, the default value is true (.T.).

<cUserFunction> is the name of a user-defined function that executes when the user presses a key not recognized by MEMOEDIT() and when no keys are pending in the keyboard buffer. <cUserFunction> is specified as a character value without parentheses or arguments. Specifying false (.F.) for this argument displays <cString> and causes MEMOEDIT() to immediately terminate. If this argument is specified, the automatic behavior of MEMOEDIT() changes. Refer to the discussion below.

<nLineLength> determines the length of lines displayed in the MEMOEDIT() window. If a line is greater than <nLineLength>, it is word wrapped to the next line in the MEMOEDIT() window. If <nLineLength> is greater than the number of columns in the MEMOEDIT() window, the window will scroll if the cursor moves past the window border. If <nLineLength> is not specified, the default line length is (<nRight> - <nLeft>).

<nTabSize> determines the size of a tab character to insert when the user presses Tab. If <nTabSize> is not specified, four spaces are inserted instead of a tab character.

<nTextBufferRow> and **<nTextBufferColumn>** define the display position of the cursor within the text buffer when MEMOEDIT() is invoked. <nTextBufferRow> begins with one and <nTextBufferColumn> begins with zero. If these arguments are not specified, the cursor is placed at row one and column zero of the MEMOEDIT() window.

<nWindowRow> and **<nWindowColumn>** define the initial position the cursor within the MEMOEDIT() window. Row and column positions begin with zero. If these arguments are not specified, the initial window position is row zero and the current cursor column position.

Returns

MEMOEDIT() returns the text buffer if the user terminates editing with

Ctrl-W or a copy of <cString> if user terminates with Esc.

Description

MEMOEDIT() is a user interface and general purpose text editing function that edits memo fields and long character strings. Editing occurs within a specified window region placed anywhere on the screen. Like the other user interface functions (ACHOICE() and DBEDIT()), MEMOEDIT() supports a number of different modes and includes a user function that allows key reconfiguration and other activities relevant to programming the current text editing task.

. **The text buffer:** When you invoke MEMOEDIT() and specify <cString>, it is copied to the text buffer. The user actually edits the text buffer. If the <cString> is not specified, the user is presented with an empty text buffer to edit.

  When the user exits MEMOEDIT() by pressing Ctrl-W, the contents of the text buffer are returned. If the user exits by pressing Esc, the text buffer is discarded and the original <cString> value is returned. In either case, the return value can then be assigned to a variable or memo field, or passed as an argument to another function.

. **Editing modes:** MEMOEDIT() supports two editing modes depending on the value of <lEditMode>. When <lEditMode> is true (.T.), MEMOEDIT() enters edit mode and the user can change the contents of the MEMOEDIT() text buffer. When <lEditMode> is false (.F.), MEMOEDIT() enters browse mode and the user can only navigate about the text buffer but cannot edit or insert new text. To make browsing easier for the user, the scrolling is disabled so Uparrow and Dnarrow scroll the text buffer up or down one line within the MEMOEDIT() window.

. **Entering and editing text:** Within MEMOEDIT(), the user can enter and edit text by positioning the cursor, adding, or deleting characters. To facilitate editing the text, there are a number of different navigation and editing keys:

  MEMOEDIT() Navigation and Editing Keys

Key	Action
Uparrow/Ctrl-E	Move up one line
Dnarrow/Ctrl-X	Move down one line
Leftarrow/Ctrl-S	Move left one character
Rightarrow/Ctrl-D	Move right one character
Ctrl-Leftarrow/Ctrl-A	Move left one word
Ctrl-Rightarrow/Ctrl-F	Move right one word
Home	Move to beginning of current line
End	Move to end of current line
Ctrl-Home	Move to beginning of current window
Ctrl-End	Move to end of current window
PgUp	Move to previous edit window
PgDn	Move to next edit window
Ctrl-PgUp	Move to beginning of memo
Ctrl-PgDn	Move to end of memo
Return	Move to beginning of next line
Delete	Delete character at cursor
Backspace	Delete character to left of cursor
Tab	Insert tab character or spaces
Printable characters	Insert character
Ctrl-Y	Delete the current line
Ctrl-T	Delete word right
Ctrl-B	Reform paragraph
Ctrl-V/Ins	Toggle insert mode
Ctrl-W	Finish editing with save
Esc	Abort edit and return original

When the user is entering text, there are two text entry modes, insert and overstrike. When MEMOEDIT() is invoked, the default mode is overstrike. Edit mode changes in MEMOEDIT() when the user presses Ins which toggles between the insert and overstrike. It also changes in a user function using READINSERT() or RETURNing 22. In insert mode, characters are entered into the text buffer at the current cursor position and the remainder of the text moves to the right. Insert mode is indicated in the scoreboard area. In overstrike mode, characters are entered at the current cursor position overwriting existing characters while the rest of the text buffer remains in its current position.

As the user enters text and the cursor reaches the edge of the MEMOEDIT() window, the current line wraps to the next line in the text buffer and a soft carriage return (CHR(141)) is inserted into the text. If the <nLineLength> argument is specified, text wraps when the cursor position is the same as <nLineLength>. If <nLineLength> is greater than the width of the MEMOEDIT() window, the window scrolls. To explicitly start a new line or paragraph, the user must press Return.

. **The edit screen:** When MEMOEDIT() displays, it overwrites the specified area of the screen and does not save the underlying screen. Additionally, it does not display a border or a title. To provide these facilities, you must create a procedure or user-defined function that performs these actions then calls MEMOEDIT(). See the example below.

. **The user function:** <cUserFunction>, a user-defined function specified as an argument, handles key exceptions and reconfigures special keys. The user function is called at various times by MEMOEDIT(), most often in response to keys it does not recognize. Keys that instigate a key exception are all available control keys, function keys, and Alt keys. Since these keys are not processed by MEMOEDIT(), they can be reconfigured. Some of these keys have a default action assigned to them. In the user function, you perform various actions, depending on the current MEMOEDIT() mode, then RETURN a value telling MEMOEDIT() what to do next.

When the user function argument is specified, MEMOEDIT() defines two classes of keys: nonconfigurable and key exceptions. When a nonconfigurable key is pressed, MEMOEDIT() executes it, otherwise a key exception is generated and the user function is called. When there are no keys left in the keyboard buffer for MEMOEDIT() to process, the user function is called once again.

When MEMOEDIT() calls the user function, it automatically passes three parameters indicating the MEMOEDIT() mode, the current text buffer line, and the current text buffer column. The mode indicates the current state of MEMOEDIT() depending on the last key pressed or the last action taken prior to executing the user function. The following modes are possible:

MEMOEDIT() Modes

Mode	Memoedit.ch	Description
0	ME_IDLE	Idle, all keys processed
1	ME_UNKEY	Unknown key, memo unaltered
2	ME_UNKEYX	Unknown key, memo altered
3	ME_INIT	Initialization mode

See Also: MEMOEDIT()...

A mode value of 3 indicates that MEMOEDIT() is in initialization mode. When you specify <cUserFunction>, MEMOEDIT() makes a call to the user function immediately after being invoked. At this point, you RETURN a request to set MEMOEDIT()'s various text formatting modes: word wrap, scroll, or insert. MEMOEDIT() calls the user function repeatedly, remaining in the initialization mode until you RETURN 0. The text buffer is then displayed, and the user enters the edit mode set by <lEditMode>. Note that if word wrap is on when MEMOEDIT() changes from initialization to edit mode, the entire text buffer is formatted with <nLineLength>. To prevent this initial formatting, toggle word wrap off during initialization. Note also that the toggles for scroll and word wrap are not assigned to any key, but can be assigned to a key from the user function.

Modes 1 and 2 indicate that MEMOEDIT() has fetched an unrecognizable or configurable key from the keyboard buffer. Configurable keys are processed by RETURNing 0 to execute the MEMOEDIT() default action. RETURNing a different value executes another key action, thereby redefining the key. If the key is an unrecognizable key you can define an action for it by RETURNing a value requesting a key action or perform an action of your own definition.

Mode 0 indicates that MEMOEDIT() is now idle with no more keys to process. Whenever, MEMOEDIT() becomes idle, it always make a call to the user function. At this point, you generally update line and column number displays.

The other two parameters, current line and column, indicate the current cursor position in the text buffer when the user function is called. The line parameter begins with position one, and column begins with position zero.

When the mode is either 1, 2, or 3, you can return a value instructing MEMOEDIT() what action to perform next. The following table summarizes the possible return values and their consequences:

MEMOEDIT() User Function Return Values

Value	Memoedit.ch	Action
0	ME_DEFAULT	Perform default action
1-31	ME_UNKEY	Process requested action corresponding to key value
32	ME_IGNORE	Ignore unknown key
33	ME_DATA	Treat unknown key as data
34	ME_TOGGLEWRAP	Toggle word wrap mode
35	ME_TOGGLESCROLL	Toggle scroll mode
100	ME_WORDRIGHT	Perform word-right operation
101	ME_BOTTOMRIGHT	Perform bottom-right operation

- **Header files:** To make the mode and request values easier to remember and use, the header file Memoedit.ch is supplied in \CLIPPER5\INCLUDE. Additionally Inkey.ch, which contains manifest constants for all the INKEY() values, is also located in the same directory.

Notes

- **Configuring keys:** If the <cUserFunction> is specified, the keys in the table below are configurable.

MEMOEDIT() Configurable Keys

Key	Default Action
Ctrl-Y	Delete the current line

```
Ctrl-T          Delete word right
Ctrl-B          Reform Paragraph
Ctrl-V/Ins      Toggle insert mode
Ctrl-W          Finish editing with save
Esc             Abort edit and return original
------------------------------------------------------------------------
```

If the key is configurable, RETURNing 0 executes the MEMOEDIT() default action. RETURNing a different value, however, executes another key action thereby redefining the key. If the key is not a configurable key recognized by MEMOEDIT(), you can define an action for it also by RETURNing a value requesting a key action from the table above.

. **Word wrap:** Word wrap is a formatting mode you can toggle by RETURNing 34 from the user function. When word wrap is on (the default), MEMOEDIT() inserts a soft carriage return/linefeed at the closest word break to the window border or line length, whichever occurs first. When word wrap is off, MEMOEDIT() scrolls text buffer beyond the edge of the window until the cursor reaches the end of line. At this point, the user must press Return (inserting a hard carriage return/linefeed) to advance to the next line.

. **Reforming paragraphs:** Pressing Ctrl-B or RETURNing a 2 from a user function reformats the text buffer until a hard carriage return (end of paragraph) or the end of the text buffer is reached. This happens regardless of whether word wrap is on or off.

. **Soft carriage returns:** In CA-Clipper, the insertion of soft carriage return/linefeed characters is never allowed to change the significant content of the text. That is, when a soft carriage return/linefeed is inserted between two words, the space characters between the two words are preserved. When text is reformatted, any soft carriage return/linefeed characters are removed. This leaves the text in its original form and properly handles the case where a soft carriage return/linefeed has been inserted in the middle of a word.

In the Summer '87 version of MEMOEDIT(), when a soft carriage return/linefeed is inserted, a single space character is removed from the text at that point. If the text is later reformatted using a different line width, each soft carriage return/linefeed is replaced by a single space. However, if the text string is reformatted using any of the CA-Clipper text handling functions, words that were separated by a soft carriage return/linefeed will be run together because the soft carriage return/linefeed is not replaced with a space.

To prevent this, text that was formatted using Summer '87 MEMOEDIT() should be processed to change any soft carriage return/linefeed pairs into space characters. This can be accomplished using the STRTRAN() function as follows.

```
STRTRAN( <text>, CHR(141)+CHR(10), " " )
```

To convert memo values in an existing database, the following two line program can be used:

```
USE <xcDatabase>
REPLACE ALL <idMemo> WITH ;
   STRTRAN( <idMemo>, CHR(141)+CHR(10), " " )
```

Because of the (.dbt) file format, replacing all occurrences of a memo field can cause the (.dbt) file to grow significantly. The (.dbt) file can be reduced by copying the (.dbf) to a new file.

For very large (.dbt) files, it may not be feasible to perform the above procedure. The supplied utility program, DBT50.EXE located in

\CLIPPER5\BIN, may be useful in these cases. DBT50 scans an entire (.dbt) file, replacing any soft carriage return/linefeed pairs with two spaces. Although this has the undesirable effect of causing certain words to be separated by two spaces instead of one, it allows the file to be processed in place without using additional disk space. DBT50 modifies only soft carriage return/linefeed pairs in the target file. Other text is unaffected.

- **Editing text files:** MEMOEDIT() edits text files if the text file can be read into a CA-Clipper character variable. This can be done with the MEMOREAD() function. After editing the contents of the text file held in the character variable, write it back to the file using MEMOWRIT().

Examples

- This example lets you browse a memo field but prevents any changes to the text buffer:

```
USE Customer NEW
SET CURSOR OFF
MEMOEDIT(CustNotes, 5, 10, 20, 69, .F.)
SET CURSOR ON
```

- This example allows editing of a memo field, assigning the changes back to the memo field:

```
USE Customer NEW
REPLACE CustNotes WITH ;
      MEMOEDIT(CustNotes, 5, 10, 20, 69)
```

- This example creates a character string using MEMOEDIT():

```
LOCAL cNotes
cNotes = MEMOEDIT()
```

- This example is a user-defined function that edits a character string in a boxed window displayed with a title:

```
FUNCTION EditMemo( cString, cTitle,;
         nTop, nLeft, nBottom, nRight )
   LOCAL cScreen := SAVESCREEN(nTop, nLeft,;

         nBottom, nRight)
   @ nTop - 1, nLeft - 2 CLEAR TO nBottom + 1,;
         nRight + 2
   @ nTop - 1, nLeft - 2 TO nBottom + 1, nRight + 2
   @ nTop - 1, nLeft SAY "[" + cTitle + "]"
   cString = MEMOEDIT(cString, nTop, nLeft,;
         nBottom, nRight)
   RESTSCREEN(nTop, nLeft, nBottom, nRight, cScreen)
   RETURN (cString)
```

- This example reads the contents of a text file into a character variable, edits it, then writes it back to disk:

```
LOCAL cString := MEMOREAD("Text.txt")
cString := MEMOEDIT(cString)
IF !MEMOWRIT("Text.txt", cString)
   ? "Write error"
   BREAK
ENDIF
RETURN
```

- This example contains a user-defined function that displays a message describing the current MEMOEDIT() mode. Additionally, while in ME_UNKEY mode, the function will perform either a ME_WORDRIGHT or ME_BOTTOMRIGHT action depending on which associated function key is

pressed:

```
#include "memoedit.ch"
#include "inkey.ch"

PROCEDURE Main()
   USE Customer NEW
   REPLACE CustNotes WITH;
   MEMOEDIT( CustNotes, 5, 5, 15, 75, .T., "MemoUDF" )
RETURN

FUNCTION MemoUDF( nMode, nLine, nCol )
   LOCAL nKey := LASTKEY()
   LOCAL nRetVal := ME_DEFAULT         // Default return action

   DO CASE
   CASE nMode == ME_IDLE
      @ 20, 5 SAY "MemoMode is ME_IDLE  "
   CASE nMode == ME_UNKEY
      @ 20, 5 SAY "MemoMode is ME_UNKEY "
      DO CASE
      CASE nKey == K_F2
         nRetVal := ME_WORDRIGHT
      CASE nKey == K_F3
         nRetVal := ME_BOTTOMRIGHT
      ENDCASE
      CASE nMode == ME_UNKEYX
      @ 20, 5 SAY "MemoMode is ME_UNKEYX"
   OTHERWISE

      @ 20, 5 SAY "MemoMode is ME_INIT  "
   ENDCASE

RETURN nRetVal
```

Files: Library is EXTEND.LIB, header files are Memoedit.ch and Inkey.ch.

See Also: LASTKEY() MEMOREAD() MEMOTRAN() MEMOWRIT()

MEMOLINE()
Extract a line of text from a character string or memo field

Syntax

MEMOLINE(<cString>,
[<nLineLength>],
[<nLineNumber>],
[<nTabSize>],
[<lWrap>]) --> cLine

Arguments

<cString> is the memo field or character string from which to extract a line of text.

<nLineLength> specifies the number of characters per line and can be between four and 254. If not specified, the default line length is 79.

<nLineNumber> is the line number to extract. If not specified, the default value is one.

<nTabSize> defines the tab size. If not specified, the default value is four. If <nTabSize> is greater than or equal to <nLineLength>, then the tab size is automatically converted to <nLineLength> - 1.

<lWrap> toggles word wrap on and off. Specifying true (.T.) toggles word wrap on; false (.F.) toggles it off. If not specified, the default value is true (.T.).

Returns

MEMOLINE() returns the line of text specified by <nLineNumber> in <cString> as a character string. If the line has fewer characters than the indicated length, the return value is padded with blanks. If the line number is greater than the total number of lines in <cString>, MEMOLINE() returns a null string ("").

If <lWrap> is true (.T.) and the indicated line length breaks the line in the middle of a word, that word is not included as part of the return value but shows up at the beginning of the next line extracted with MEMOLINE().

If <lWrap> is false (.F.), MEMOLINE() returns only the number of characters specified by the line length. The next line extracted by MEMOLINE() begins with the character following the next hard carriage return, and all intervening characters are not processed.

Description

MEMOLINE() is a memo function used with MLCOUNT() to extract lines of text from character strings and memo fields based on the number of characters per line. It is the most basic facility provided by CA-Clipper to display memo fields and long strings.

The basic method of operation is to determine the number of lines in the memo field or character string using MLCOUNT() with the same number of characters per line, tab size, and wrapping behavior as you intend to use with MEMOLINE(). Using this value as the upper boundary of a FOR...NEXT, each line of the memo field or character string can be extracted with MEMOLINE() and processed with any combination of output commands and functions required.

Examples

. This example demonstrates the general method for displaying memo fields and long character strings using the combination of

```
MLCOUNT() and MEMOLINE():

LOCAL nLineLength := 40, nTabSize := 3, lWrap := .T.
LOCAL nLines, nCurrentLine
USE Customer INDEX CustName NEW
//
nLines := MLCOUNT(CustNotes, nLineLength,;
      nTabSize, lWrap)
//
SET PRINTER ON
FOR nCurrentLine := 1 TO nLines
   ? MEMOLINE(CustNotes, nLineLength, nCurrentLine,;
         nTabSize, lWrap)
NEXT
SET PRINTER OFF
```

Files: Library is EXTEND.LIB.

See Also: MEMOEDIT() MLCOUNT() MLPOS()

MEMOREAD()
Return the contents of a disk file as a character string

Syntax

MEMOREAD(<cFile>) --> cString

Arguments

<cFile> is the name of the file to read from disk. It must include an extension if there is one, and can optionally include a path.

Returns

MEMOREAD() returns the contents of a text file as a character string. The maximum file size that can be read is 65,535 characters (64K)--the maximum size of a character string. If <cFile> cannot be found, MEMOREAD() returns a null string ("").

Description

MEMOREAD() is a memo function that reads a disk file into memory where it can be manipulated as a character string or assigned to a memo field. MEMOREAD() is used with MEMOEDIT() and MEMOWRIT() to edit an imported disk file then write it back to disk. MEMOREAD() searches for <cFile> beginning with the current DOS directory. If the file is not found, MEMOREAD() searches the DOS path. MEMOREAD() does not use the CA-Clipper DEFAULT or PATH to search for <cFile>.

In a network environment, MEMOREAD() attempts to open the specified file shared and read-only. If the file is opened exclusive by another process, MEMOREAD() returns a null string ("").

Examples

. This example uses MEMOREAD() to assign the contents of a text file to the Notes memo field and to a character variable:

```
REPLACE Notes WITH MEMOREAD("Temp.txt")
cString = MEMOREAD("Temp.txt")
```

. This example defines a function that edits a disk file:

```
FUNCTION Editor( cFile )
   LOCAL cString
   IF (cString := MEMOREAD(cFile)) == ""
      ? "Error reading " + cFile
      RETURN .F.
   ELSE
      MEMOWRIT(cFile, MEMOEDIT(cString))
      RETURN .T.
   ENDIF
```

Files: Library is EXTEND.LIB.

See Also: MEMOEDIT() MEMOWRIT() REPLACE

MEMORY()
Determine the amount of available free pool memory

Syntax

MEMORY(<nExp>) --> nKbytes

Arguments

<nExp> is a numeric value that determines the type of value MEMORY() returns as follows:

MEMORY() Argument Values

Value	Meaning
0	Estimated total space available for character values
1	Largest contiguous block available for character values
2	Area available for RUN commands

Returns

MEMORY() returns an integer numeric value representing the amount of memory available, in one-kilobyte increments.

Description

MEMORY() is an environment function that reports various states of free pool memory. (Free pool is the dynamic region of memory that stores character strings and executes RUN commands.)

Examples

. This example uses MEMORY() before a RUN command to determine if there is enough memory available to execute the external program:

```
#define MEM_CHAR   0
#define MEM_BLOCK  1
#define MEM_RUN    2
//
IF MEMORY(MEM_RUN) >= 128
   RUN MYPROG
ELSE
   ? "Not enough memory to RUN"
   BREAK
ENDIF
```

Files: Library is CLIPPER.LIB.

MEMOTRAN()
Replace carriage return/linefeeds in character strings

Syntax

```
MEMOTRAN(<cString>,
   [<cReplaceHardCR>],
   [<cReplaceSoftCR>]) --> cNewString
```

Arguments

<cString> is the character string or memo field to search.

<cReplaceHardCR> is the character to replace a hard carriage return/linefeed pair with. If not specified, the default value is a semicolon (;).

<cReplaceSoftCR> is the character to replace a soft carriage return/linefeed pair with. If not specified, the default value is a space.

Returns

MEMOTRAN() returns a copy of <cString> with the specified carriage return/linefeed pairs replaced.

Description

MEMOTRAN() is a memo function that converts a memo field or long character string containing hard and soft carriage return/linefeed characters into a form that can be displayed. These two character combinations are end of line formatting indicators placed in the string by MEMOEDIT(). Soft carriage returns (CHR(141)) are inserted when a line longer than the width of the MEMOEDIT() window wraps. Hard carriage returns (CHR(13)) are inserted when the user explicitly presses Return.

MEMOTRAN() is particularly useful when displaying a memo field in a REPORT FORM which does not wrap when a soft carriage return is encountered. MEMOTRAN() resolves this by converting soft carriage returns to spaces. Note, however, that you must declare MEMOTRAN() as external using the REQUEST statement if it is used in a REPORT FORM and not specified anywhere else in the current program.

Examples

. This example strips all end of line characters from a memo field:

```
REPLACE Notes WITH MEMOTRAN(Notes)
```

Files: Library is EXTEND.LIB.

See Also: EXTERNAL* HARDCR() REPORT FORM REQUEST STRTRAN()

MEMOWRIT()
Write a character string or memo field to a disk file

Syntax

MEMOWRIT(<cFile>, <cString>) --> lSuccess

Arguments

<cFile> is the name of the target disk file including the file extension and optional path and drive designator.

<cString> is the character string or memo field to write to <cFile>.

Returns

MEMOWRIT() returns true (.T.) if the writing operation is successful; otherwise, it returns false (.F.).

Description

MEMOWRIT() is a memo function that writes a character string or memo field to a disk file. If a path is not specified, MEMOWRIT() writes <cFile> to the current DOS directory and not the current DEFAULT directory. If <cFile> already exists, it is overwritten.

MEMOWRIT() is generally used with MEMOREAD() to load text files into memory where they can be edited, displayed, and written back to disk. You can also use MEMOWRIT() as a quick way of exporting a memo field to a text file.

Examples

. This example uses MEMOWRIT() with MEMOREAD() to allow editing of memo fields with an external editor:

```
LOCAL cEditor := "MYEDIT.EXE"
USE Sales NEW
IF MEMOWRIT("Cliptmp.txt", Notes)
   RUN (cEditor + " Cliptmp.txt")
   REPLACE Notes WITH MEMOREAD("Cliptmp.txt")
ELSE
   ? "Error while writing Cliptmp.txt"
   BREAK
ENDIF
```

Files: Library is EXTEND.LIB.

See Also: MEMOEDIT() MEMOREAD()

MEMVARBLOCK()
Return a set-get code block for a given memory variable

Syntax

MEMVARBLOCK(<cMemvarName>) --> bMemvarBlock

Arguments

<cMemvarName> is the name of the variable referred to by the set-get block, specified as a character string.

Returns

MEMVARBLOCK() returns a code block that when evaluated sets (assigns) or gets (retrieves) the value of the given memory variable. If <cMemvarName> does not exist, MEMVARBLOCK() returns NIL.

Description

The code block created by MEMVARBLOCK() has two operations depending on whether an argument is passed to the code block when it is evaluated. If evaluated with an argument, it assigns the value of the argument to <cMemvarName>. If evaluated without an argument, the code block retrieves the value of <cMemvarName>.

Notes

. MEMVARBLOCK() creates set-get blocks only for variables whose names are known at runtime. MEMVARBLOCK(), therefore, cannot be used to create set-get blocks for local or static variables. The same restriction applies to creating blocks using the macro operator (&).

Examples

. This example compares MEMVARBLOCK() to a code block created using the macro operator (&). Note that using MEMVARBLOCK() allows you to avoid the speed and size overhead of the macro operator:

```
PRIVATE var := "This is a string"
//
// Set-Get block defined using macro operator
bSetGet := &( "{ |setVal|;
      IF( setVal == NIL, var, var := setVal ) }" )
// Set-Get block defined using MEMVARBLOCK()

// bSetGet created here is the functional
// equivalent of bSetGet above
bSetGet := MEMVARBLOCK("var")
```

Files: Library is CLIPPER.LIB.

See Also: FIELDBLOCK() FIELDWBLOCK()

MIN()
Return the smaller of two numeric or date values

Syntax

MIN(<nExp1>, <nExp2>) --> nSmaller
MIN(<dExp1>, <dExp2>) --> dSmaller

Arguments

<nExp1> and **<nExp2>** are the numeric values to compare.

<dExp1> and **<dExp2>** are the date values to compare.

Returns

MIN() returns the smaller of the two arguments. The value returned is the same data type as the arguments.

Description

MIN() is a numeric and a date function that ensures the value of an expression is smaller than a specified minimum. The inverse of MIN() is MAX() which returns the greater of two numeric or date values.

Examples

. In these examples MIN() returns the smaller of two numeric values:

```
? MIN(99, 100)                  // Result: 99
? MIN(100, 99)                  // Result: 99
```

. In these examples MIN() compares date values:

```
? DATE()                        // Result: 09/01/90
? MIN(DATE(), DATE() + 30)      // Result: 09/01/90
```

Files: Library is CLIPPER.LIB.

See Also: MAX()

MLCOUNT()
Count the number of lines in a character string or memo field

---

Syntax

MLCOUNT(<cString>, [<nLineLength>],
[<nTabSize>], [<lWrap>]) --> nLines

Arguments

<cString> is the character string or memo field to count.

<nLineLength> specifies the number of characters per line and can range from four to 254. If not specified, the default line length is 79.

<nTabSize> defines the tab size. If not specified, the default value is four. If <nTabSize> is greater than or equal to <nLineLength>, then the tab size is automatically converted to <nLineLength> - 1.

<lWrap> toggles word wrap on and off. Specifying true (.T.) toggles word wrap on; false (.F.) toggles it off. If not specified, the default value is true (.T.).

Returns

MLCOUNT() returns the number of lines in <cString> depending on the <nLineLength>, the <nTabSize>, and whether word wrapping is on or off.

Description

MLCOUNT() is a memo function used with MEMOLINE() to print character strings and memo fields based on the number of characters per line. In the basic operation, use MLCOUNT() to return the number of lines in the character string or memo field. Then, using MEMOLINE() to extract each line, loop through the memo field until there are no lines left.

If <lWrap> is true (.T.) and an end of line position breaks a word, it is word wrapped to the next line and the next line begins with that word. If <lWrap> is false (.F.), MLCOUNT() counts the number of characters specified by <nLineLength> as the current line. The next line begins with the character following the next hard or soft carriage return. Intervening characters are ignored.

Examples

. This example displays the contents of each Notes memo field in the Sales database file, one line at a time:

```
USE Sales NEW
nLineLength = 65
//
DO WHILE !EOF()
   nLines = MLCOUNT(Sales->Notes, nLineLength)
   FOR nCurrLine = 1 TO nLines
      ? MEMOLINE(Sales->Notes, nLineLength, nCurrLine)
   NEXT
   SKIP
   ?
ENDDO
```

Files: Library is EXTEND.LIB.

See Also: MEMOLINE() MEMOTRAN() MLPOS()

MLCTOPOS()
Return byte position of a formatted string based on line and column position

---

Syntax

MLCTOPOS(<cText>, <nWidth>, <nLine>,
<nCol>, [<nTabSize>], [<lWrap>]) --> nPosition

Arguments

<cText> is the text string to scan.

<nWidth> is the line length formatting width.

<nLine> is the line number counting from 1.

<nCol> is the column number counting from 0.

<nTabSize> is the number of columns between tab stops. If not specified, the default is 4.

<lWrap> is the word wrap flag. If not specified, the default is true (.T.).

Returns

MLCTOPOS() returns the byte position within <cText> counting from 1.

Description

MLCTOPOS() is a memo function that determines the byte position that corresponds to a particular line and column within the formatted text. Note that the line number is one-relative, the column number is zero-relative. This is compatible with MEMOEDIT(). The return value is one-relative, making it suitable for use in SUBSTR() or other string functions.

MLCTOPOS() is used with MPOSTOLC() to create search routines or other text processing for MEMOEDIT(). Refer to the source code for the program editor (PE) found in \CLIPPER5\SOURCE\PE directory.

Examples

. This example determines the byte position of line 5, column 3 in the cText string:

```
cText := "Note the side on which the bread ;
            is buttered."
//
? MLCTOPOS(cText, 5, 3, 0)         // Result: 10
```

Files: Library is CLIPPER.LIB.

See Also: MEMOEDIT() MLPOS() MPOSTOLC()

MLPOS()
Determine the position of a line in a character string or memo field

---

Syntax

MLPOS(<cString>, <nLineLength>,
<nLine>, [<nTabSize>], [<lWrap>]) --> nPosition

Arguments

<cString> is a character string or memo field.

<nLineLength> specifies the number of characters per line.

<nLine> specifies the line number.

<nTabSize> defines the tab size. The default is four. If <nTabSize> is greater than or equal to <nLineLength>, then the tab size is adjusted to <nLineLength> - 1.

<lWrap> toggles word wrap on and off. Specifying true (.T.) toggles word wrap on, and false (.F.) toggles it off. The default is true (.T.).

Returns

MLPOS() returns the character position of <nLine> in <cString> as an integer numeric value. If <nLine> is greater than the number of lines in <cString>, MLPOS() returns the length of <cString>.

Examples

. This example uses MLPOS() to find the position of a specific line, given a line length:

```
cString = MEMOREAD("Temp.txt")
nLineLength = 40
nLine = 5
nPosition = MLPOS(cString, nLineLength, nLine)
? SUBSTR(cString, nPosition, 12)
```

Files: Library is EXTEND.LIB.

See Also: MEMOLINE() MEMOTRAN() MLCOUNT()

MOD()*
Return dBASE III PLUS modulus of two numbers

Syntax

MOD(<nDividend>, <nDivisor>) --> nRemainder

Arguments

<nDividend> is the dividend of the division operation.

<nDivisor> is the divisor of the division operation.

Returns

MOD() returns a number representing the remainder of <nDividend> divided by <nDivisor>.

Description

MOD() is a numeric function that emulates the dBASE III PLUS MOD() function. It is implemented using the CA-Clipper modulus operator (%). Note that there are differences between the dBASE III PLUS MOD() function and the CA-Clipper modulus operator which are described in the following table:

Differences Between dBASE III PLUS MOD() Function and the CA-Clipper Modulus Operator

Dividend	Divisor	Modulus Operator	MOD()	dBASE III PLUS MOD() function
3	0	Error	Error	3
3	-2	1	-1	-1
-3	2	-1	1	1
-3	0	Error	Error	-3
-1	3	-1	2	2
-2	3	-2	1	1
2	-3	2	-1	-1
1	-3	1	-2	-2

MOD() is supplied as a compatibility function and therefore not recommended. It is superseded entirely by the modulus operator (%).

Notes

- **Zero divisor in dBASE III PLUS:** In dBASE III PLUS, a zero divisor returns the dividend for every value of the dividend. In CA-Clipper, by contrast, the modulus of any dividend using a zero divisor causes a runtime error.

- **Zero divisor in earlier versions:** In versions of CA-Clipper prior to Summer '87, a modulus operation with a zero divisor returned zero for all dividends. In Summer '87 and later versions, it returns a runtime error.

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\MOD.PRG.

See Also: %

MONTH()
Convert a date value to the number of the month

Syntax

MONTH(<dDate>) --> nMonth

Arguments

<dDate> is the date value to convert.

Returns

MONTH() returns an integer numeric value in the range of zero to 12. Specifying a null date (CTOD("")) returns zero.

Description

MONTH() is a date conversion function that is useful when you require a numeric month value during calculations for such things as periodic reports. MONTH() is a member of a group of functions that return components of a date value as numeric values. The group includes DAY() and YEAR() to return the day and year values as numerics. CMONTH() is a related function that allows you to return the name of the month from a date value.

Examples

. These examples return the month of the system date:

```
? DATE()                      // Result: 09/01/90
? MONTH(DATE())               // Result: 9
? MONTH(DATE()) + 1           // Result: 10
```

. This example demonstrates MONTH() acting on a null date:

```
#define NULL_DATE   (CTOD(""))
? MONTH(NULL_DATE)            // Result: 0
```

Files: Library is CLIPPER.LIB.

See Also: CMONTH() DAY() DOW() YEAR()

MPOSTOLC()
Return line and column position of a formatted string based on a specified byte position

---

Syntax

MPOSTOLC(<cText>, <nWidth>, <nPos>,
[<nTabSize>], [<lWrap>]) --> aLineColumn

Arguments

<cText> is a text string.

<nWidth> is the length of the formatted line.

<nPos> is the byte position within text counting from one.

<nTabSize> is the number of columns between tab stops. If not specified, the default is four.

<lWrap> is the word wrap flag. If not specified, the default is true (.T.).

Returns

MPOSTOLC() returns an array containing the line and the column values for the specified byte position, <nPos>.

Description

MPOSTOLC() is a memo function that determines the formatted line and column corresponding to a particular byte position within <cText>. Note that the line number returned is one-relative, the column number is zero-relative. This is compatible with MEMOEDIT(). <nPos> is one-relative, compatible with AT(), RAT(), and other string functions.

MPOSTOLC(), used with MLCTOPOS(), can create search routines or other text processing for MEMOEDIT(). Refer to the source code for the program editor (PE) found in \CLIPPER5\SOURCE\PE directory.

Examples

. This example determines, for the text string shown, the line and column corresponding to the tenth character of the text, assuming a formatting width of five columns. A formatting width of five would cause each of the first three words to be placed on a line by itself. The tenth character of the text is the "s" in "side." The word "side" would be at the leftmost column of the third line of the formatted text, so the return value is {3, 0}:

```
cText := "Note the side on which the bread ;
      is buttered."
//
aLC := MPOSTOLC(cText, 5, 10)         // Result: {3, 0}
```

Files: Library is CLIPPER.LIB.

See Also: MEMOEDIT() MLCTOPOS() MLPOS()

NETERR()
Determine if a network command has failed

Syntax

NETERR([<lNewError>]) --> lError

Arguments

<lNewError> if specified sets the value returned by NETERR() to the specified status. <lNewError> can be either true (.T.) or false (.F.). Setting NETERR() to a specified value allows the runtime error handler to control the way certain file errors are handled. For more information, refer to Errorsys.prg.

Returns

NETERR() returns true (.T.) if a USE or APPEND BLANK fails. The initial value of NETERR() is false (.F.). If the current process is not running under a network operating system, NETERR() always returns false (.F.).

Description

NETERR() is a network function. It is a global flag set by USE, USE...EXCLUSIVE, and APPEND BLANK in a network environment. It is used to test whether any of these commands have failed by returning true (.T.) in the following situations:

NETERR() Causes

Command	Cause
USE	USE EXCLUSIVE by another process
USE...EXCLUSIVE	USE EXCLUSIVE or USE by another process
APPEND BLANK	FLOCK() or RLOCK() of LASTREC() + 1 by another user

NETERR() is generally applied in a program by testing it following a USE or APPEND BLANK command. If it returns false (.F.), you can perform the next operation. If the command is USE, you can open index files. If it is APPEND BLANK, you can assign values to the new record with a REPLACE or @...GET command. Otherwise, you must handle the error by either retrying the USE or APPEND BLANK, or terminating the current operation with a BREAK or RETURN.

Examples

. This example demonstrates typical usage of NETERR(). If the USE succeeds, the index files are opened and processing continues. If the USE fails, a message displays and control returns to the nearest BEGIN SEQUENCE construct:

```
USE Customer SHARED NEW
IF !NETERR()
   SET INDEX TO CustNum, CustOrders, CustZip
ELSE
   ? "File is in use by another"
   BREAK
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: APPEND BLANK FLOCK() RLOCK() USE

NETNAME()
Return the current workstation identification

Syntax

NETNAME() --> cWorkstationName

Returns

NETNAME() returns the workstation identification as a character string up to 15 characters in length. If the workstation identification was never set or the application is not operating under the IBM PC Network, it returns a null string ("").

Examples

. This example demonstrates the NETNAME() result when a workstation is started as a network node with a station identifier of "STATION 1":

```
? NETNAME()               // Result: STATION 1
```

. This example demonstrates the NETNAME() result when a workstation is started as a stand-alone unit:

```
? NETNAME()               // Result: ""
```

Files: Library is CLIPPER.LIB.

NEXTKEY()
Read the pending key in the keyboard buffer

Syntax

NEXTKEY() --> nInkeyCode

Returns

NEXTKEY() returns an integer numeric value ranging from -39 to 386. If the keyboard buffer is empty, NEXTKEY() returns zero. If SET TYPEAHEAD is zero, NEXTKEY() always returns zero.

Description

NEXTKEY() is a keyboard function that reads the keystroke pending in the keyboard buffer without removing it. The value returned is the INKEY() code of the key pressed--the same value as returned by INKEY() and LASTKEY(). NEXTKEY() returns values for all ASCII characters as well as function, Alt-function, Ctrl-function, Alt-letter, and Ctrl-letter key combinations.

NEXTKEY() is like the INKEY() function, but differs in one fundamental respect. INKEY() removes the pending key from the keyboard buffer and updates LASTKEY() with the value of the key. NEXTKEY(), by contrast, reads, but does not remove the key from the keyboard buffer and does not update LASTKEY().

Since NEXTKEY() does not remove the key from the keyboard buffer, it can be used to poll the keyboard then pass control to a routine that uses a wait state or INKEY() to actually fetch the key from the buffer.

For a complete list of INKEY() codes and Inkey.ch constants, refer to the CA-Clipper Inkey Codes Appendix in the Error Messages and Appendices guide.

Examples

. This example places an Esc key in the keyboard buffer then shows the differences between INKEY(), LASTKEY(), and NEXTKEY():

```
#include "Inkey.ch"
//
CLEAR TYPEAHEAD
KEYBOARD CHR(K_ESC)
//
? NEXTKEY(), LASTKEY()            // Result: 27 0
? INKEY(), LASTKEY()              // Result: 27 27
? NEXTKEY()                       // Result: 0
```

Files: Library is EXTEND.LIB, header file is Inkey.ch.

See Also: INKEY() KEYBOARD LASTKEY() SET TYPEAHEAD Inkey codes

NOSNOW()
Toggle snow suppression

Syntax

NOSNOW(<lToggle>) --> NIL

Arguments

<lToggle> is a logical value that toggles the current state of snow suppression. A value of true (.T.) enables the snow suppression on, while a value of false (.F.) disables snow suppression.

Description

NOSNOW() is used to suppress snow on CGA monitors. Typically, use NOSNOW() in the configuration section of your application to give the user the option to suppress snow.

Files: Library is CLIPPER.LIB

OS()
Return the operating system name

Syntax

OS() --> cOsName

Returns

OS() returns the operating system name as a character string.

Description

OS() is an environment function that returns the name of the disk operating system under which the current workstation is operating. The name is returned in the form of the operating system name followed by the version number.

Examples

. This example uses OS() to report the operating system the current workstation is running:

```
? OS()                  // Result: DOS 3.3
```

Files: Library is CLIPPER.LIB.

See Also: GETENV() VERSION()

OUTERR()
Write a list of values to the standard error device

Syntax

OUTERR(<exp list>) --> NIL

Arguments

<exp list> is a list of values to display and can consist of any combination of data types including memo.

Returns

OUTERR() always returns NIL.

Description

OUTERR() is identical to OUTSTD() except that it writes to the standard error device rather than the standard output device. Output sent to the standard error device bypasses the CA-Clipper console and output devices as well as any DOS redirection. It is typically used to log error messages in a manner that will not interfere with the standard screen or printer output.

Examples

. This example displays an error message along with the date and time of occurrence to the screen:

```
OUTERR("File lock failure", DATE(), TIME())
```

Files: Library is CLIPPER.LIB.

See Also: DISPOUT() OUTSTD()

OUTSTD()
Write a list of values to the standard output device

Syntax

OUTSTD(<exp list>) --> NIL

Arguments

<exp list> is a list of values to display and can consist of any combination of data types including memo.

Returns

OUTSTD() always returns NIL.

Description

OUTSTD() is a simple output function similar to QOUT(), except that it writes to the STDOUT device (instead of to the CA-Clipper console output stream). Programs with very simple output requirements (i.e., that perform no full-screen input or output) can use this function to avoid loading the terminal output subsystems. The header file Simpleio.ch redefines the ? and ?? commands to use the OUTSTD() function.

Since OUTSTD() sends its output to the standard output device, the output can be redirected using the DOS redirection symbols (>, >, |). This lets you redirect output from a CA-Clipper program to a file or pipe. Refer to your PC/MS-DOS documentation for more information on this operating system facility.

Examples

. This example uses OUTSTD() to display a list of expressions:

```
OUTSTD(Name, PADR(RTRIM(City) + "," + ;
      State, 20), ZipCode)
```

. This example redirects the output of a CA-Clipper program to a new file using the DOS redirection operator (>):

```
C>MYPROG > FILE.TXT
```

Files: Library is CLIPPER.LIB, header file is Simpleio.ch.

See Also: DISPOUT() OUTERR() QOUT()

PAD()
Pad character, date, and numeric values with a fill character

---

Syntax

```
PADL(<exp>, <nLength>, [<cFillChar>])
   --> cPaddedString
PADC(<exp>, <nLength>, [<cFillChar>])
   --> cPaddedString
PADR(<exp>, <nLength>, [<cFillChar>])
   --> cPaddedString
```

Arguments

<exp> is a character, numeric, or date value to pad with a fill character.

<nLength> is the length of the character string to return.

<cFillChar> is the character to pad <exp> with. If not specified, the default is a space character.

Returns

PADC(), PADL(), and PADR() return the result of <exp> as a character string padded with <cFillChar> to a total length of <nLength>.

Description

PADC(), PADL(), and PADR() are character functions that pad character, date, and numeric values with a fill character to create a new character string of a specified length. PADC() centers <exp> within <nLength> adding fill characters to the left and right sides; PADL() adds fill characters on the left side; and PADR() adds fill characters on the right side. If the length of <exp> exceeds <nLength>, all of the PAD() functions truncate cPaddedString to <nLength>.

PADC(), PADL(), and PADR() display variable length strings within a fixed length area. They can be used, for instance, to assure alignment with consecutive ?? commands. Another use is to display text to a fixed-width screen area assuring that previous text is completely overwritten.

PADC(), PADL(), and PADR() are the inverse of the ALLTRIM(), RTRIM(), and LTRIM() functions which trim leading and trailing space from character strings.

Examples

. This example uses PADR() to format a record number display on a status line filling the allocated space:

```
IF EOF()
   @ 23, 45 PADR("EOF/" + LTRIM(STR(LASTREC())), 20)
ELSEIF BOF()
   @ 23, 45 PADR("BOF/" + LTRIM(STR(LASTREC())), 20)
ELSE
   @ 23, 45 SAY PADR("Record " + LTRIM(STR(RECNO()) ;
         + "/" + LTRIM(STR(LASTREC())), 20)
ENDIF
```

Files: Library is EXTEND.LIB.

See Also: ALLTRIM() LTRIM() RTRIM()

PCOL()
Return the current column position of the printhead

Syntax

PCOL() --> nColumn

Returns

PCOL() returns an integer numeric value representing the last printed column position, plus one. The beginning column position is zero.

Description

PCOL() is a printer function that reports the column position of the printhead after the last print operation. PCOL() is updated only if either SET DEVICE TO PRINTER or SET PRINTER ON is in effect. PCOL() is the same as COL() except that it relates to the printer rather than the screen. PCOL() is updated in the following ways:

- Application startup sets PCOL() to zero
- EJECT resets PCOL() to zero
- A print operation sets PCOL() to the last column print position plus one
- SETPRC() sets PCOL() to the specified column position

PCOL(), used with PROW(), prints a value relative to the last value printed on the same line. This makes it easier to align columns when printing a columnar report. A value is printed in the next column by specifying its position as PCOL() + <column offset>. Note that PCOL() is effective for alignment only if the column values are fixed-width. To guarantee fixed-width column values, format the output using TRANSFORM(), the PICTURE clause of @...SAY, or any of the PAD() functions.

Notes

- **Printer control codes:** Sending control codes to the printer causes PCOL() to become out of sync with the printhead position. Although control codes do not print, this discrepancy happens because CA-Clipper counts all characters sent to the printer regardless of how the printer treats them. To make the necessary adjustment, save the current PROW() and PCOL() values, send the control codes, then use SETPRC() to restore the original PROW() and PCOL() values.
- **SET MARGIN:** PCOL() cannot reliably be used with SET MARGIN to print with @...SAY. When printing with @...SAY, the current MARGIN value is always added to the specified column position before output is sent to the printer. This effectively adds the MARGIN value to PCOL() for each invocation of @...SAY to the same print line.

Examples

- In this example, PCOL() creates a simple report that prints a listing of Customer names, addresses, and phone numbers:

```
LOCAL nLine := 99, nPage := 1
USE Customer INDEX CustName NEW
SET DEVICE TO PRINTER
DO WHILE !EOF()
   IF nLine > 55
      PageTop(nPage)
      nLine := 1
      nPage++
```

```
      ENDIF
      @ nLine, 10 SAY CustName
      @ nLine, PCOL() + 2;
         SAY RTRIM(City) + ", " + RTRIM(State) + ZipCode;
               PICTURE REPLICATE("X", 35)
      @ nLine, PCOL() + 2;
         SAY Phone;
               PICTURE "@R (999) 999-9999"
      nLine++
      SKIP
   ENDDO
   SET DEVICE TO SCREEN
   CLOSE
```

Files: Library is CLIPPER.LIB.

See Also: COL() PAD() PROW() ROW() SET DEVICE SETPRC()

PCOUNT()
Determine the position of the last actual parameter passed

Syntax

PCOUNT() --> nLastArgumentPos

Returns

PCOUNT() returns, as an integer numeric value, the position of the last argument passed. If no arguments are passed, PCOUNT() returns zero.

Description

PCOUNT() reports the position of the last argument in the list of arguments passed when a procedure or user-defined function is invoked. This information is useful when determining whether arguments were left off the end of the argument list. Arguments skipped in the middle of the list are still included in the value returned.

To determine if a parameter did not receive a value, test it for NIL. Skipped parameters are uninitialized and, therefore, return NIL when accessed. Another method is to test parameters with the VALTYPE() function. This can establish whether the argument was passed and enforce the correct type at the same time. If a parameter was not supplied, a default value can be assigned.

For more information on passing parameters, refer to the Basic Concepts chapter in the Programming and Utilities guide.

Examples

. This example is a user-defined function that opens a database file and uses PCOUNT() to determine whether the calling procedure passed the name of the database file to open. If the name was not passed, OpenFile() asks for the name:

```
FUNCTION OpenFile( cFile )
   IF PCOUNT() = 0
      ACCEPT "File to use: " TO cFile
   ENDIF
   USE (cFile)
   RETURN (NETERR())
```

Files: Library is CLIPPER.LIB.

See Also: DO* FUNCTION PARAMETERS PROCEDURE VALTYPE()

PROCLINE()
Return the source line number of the current or previous activation

---

Syntax

PROCLINE([<nActivation>]) --> nSourceLine

Arguments

<nActivation> is a numeric value that specifies which activation to query. Zero refers to the current activation, one refers to the previous activation, etc. If not specified, the default value is zero.

Returns

PROCLINE() returns the line number of the last line executed in a currently executing procedure, function, or code block as an integer numeric value. If the /L compiler option suppresses line number information, PROCLINE() always returns zero.

Description

PROCLINE() queries the CA-Clipper activation stack to determine the last line executed in a currently executing procedure, user-defined function, or code block. The activation stack is an internal structure that maintains a record of each procedure, function, or code block invocation. A line number is relative to the beginning of the original source file. A line includes a comment, blank line, preprocessor directive, and a continued line. A multistatement line is counted as a single line.

For the current activation, PROCLINE() returns the number of the current line. For a previous activation, PROCLINE() returns the number of the line that invoked the procedure, or a user-defined function in which PROCLINE() is invoked.

If the activation being queried is a code block evaluation, PROCLINE() returns the line number of the procedure in which the code block was originally defined.

PROCLINE() is used with PROCNAME() to report debugging information.

Examples

. In this example, PROCLINE() returns the line number for the current activation, followed by the line number of the previous activation:

```
// First line of source file
MyFunction()
   RETURN

FUNCTION MyFunction
   ? PROCLINE()      // Result: 6 (current activation)
   ? PROCLINE(1)     // Result: 2 (previous activation)
   RETURN NIL
```

Files: Library is CLIPPER.LIB.

See Also: PROCNAME()

PROCNAME()
Return the name of the current or previous procedure or user-defined function

---

Syntax

PROCNAME([<nActivation>]) --> cProcedureName

Arguments

<nActivation> specifies which activation to query. A value of zero refers to the current activation, a value of one refers to the previous activation, etc. If unspecified, the default value is zero.

Returns

PROCNAME() returns the name of a currently executing procedure, function, or code block, as a character string.

Description

PROCNAME() queries the CA-Clipper activation stack to determine the name of a currently executing procedure, user-defined function, or code block. The activation stack is an internal structure that maintains a record of each procedure, function, or code block invocation.

For the current activation, PROCNAME() returns the name of the current procedure or user-defined function. For a previous activation, PROCNAME() returns the name of the procedure or user-defined function that invoked the current procedure.

If the activation being queried is a code block evaluation, PROCNAME() returns the name of the procedure or user-defined function that defined the code block, preceded by (b). If the activation being queried is a memvar, PROCNAME() returns the name preceded by M->.

PROCNAME() is used with PROCLINE() to report debugging information.

Examples

. This example is a user-defined function you can call during a debugging phase of program development to display the activation stack with line numbers:

```
FUNCTION ListStack( cMessage )
   LOCAL nActivation := 1
   ? cMessage
   DO WHILE !(PROCNAME(nActivation) == "")
      ? "Called from:", PROCNAME(nActivation),;
         "(" + LTRIM(STR(PROCLINE(nActivation))) + ")"
      nActivation++
   ENDDO
   QUIT
   RETURN NIL
```

Files: Library is CLIPPER.LIB.

See Also: PROCLINE()

PROW()
Return the current row position of the printhead

Syntax

PROW() --> nRow

Returns

PROW() returns an integer numeric value that represents the number of the current line sent to the printer. The beginning row position is zero.

Description

PROW() is a printer function that reports the row position of the printhead after the last print operation. PROW() is updated only if either SET DEVICE TO PRINTER or SET PRINTER ON is in effect. PROW() is like ROW() except that it relates to the printer rather than the screen. PROW() is updated in the following ways:

- Application startup sets PROW() to zero
- EJECT resets PROW() to zero
- A print operation sets PROW() to the last row print position
- SETPRC() sets PROW() to the specified row position

PROW(), used with PCOL() prints a value to a new row relative to the last row printed. If the printhead is positioned to a new row with a control code, a linefeed (CHR(10)), or formfeed (CHR(12)), PROW() is not updated and, therefore, will not return the expected value. To prevent this discrepancy, reset PROW() to the correct value with SETPRC() after sending any of these characters to the printer.

Examples

- This example uses PROW() and SETPRC() to create a simple one-across label program that prints with @...SAY instead of ?:

```
USE Customer INDEX CustName NEW
SET DEVICE TO PRINTER
SETPRC(2, 0)
DO WHILE !EOF()
   @ PROW(), 3 SAY CustName
   @ PROW() + 1, 3 SAY RTRIM(City) + ",;
          " + RTRIM(State) + ZipCode
   @ PROW() + 1, 3 SAY Phone PICTURE "@R ;
         (999) 999-9999"
   SETPRC(2, 0)
   SKIP
ENDDO
SET DEVICE TO SCREEN
CLOSE
```

Files: Library is CLIPPER.LIB.

See Also: COL() EJECT PCOL() ROW() SET DEVICE SETPRC()

QOUT()
Display a list of expressions to the console

Syntax

QOUT([<exp list>]) --> NIL
QQOUT([<exp list>]) --> NIL

Arguments

<exp list> is a comma-separated list of expressions (of any data type other than array or block) to display to the console. If no argument is specified and QOUT() is specified, a carriage return/linefeed pair is displayed. If QQOUT() is specified without arguments, nothing displays.

Returns

QOUT() and QQOUT() always return NIL.

Description

QOUT() and QQOUT() are console functions. These are the functional primitives that create the ? and ?? commands, respectively. Like the ? and ?? commands, they display the results of one or more expressions to the console. QOUT() outputs carriage return and linefeed characters before displaying the results of <exp list>. QQOUT() displays the results of <exp list> at the current ROW() and COL() position. When QOUT() and QQOUT() display to the console, ROW() and COL() are updated. If SET PRINTER is ON, PROW() and PCOL() are updated instead. If <exp list> is specified, both QOUT() and QQOUT() display a space between the results of each expression.

You can use QOUT() and QQOUT() for console display within an expression. This is particularly useful for blocks, iteration functions such as AEVAL() and DBEVAL(), and in a list of statements in the output pattern of a user-defined command definition.

Examples

. This example uses QOUT() with AEVAL() to list the contents of a literal array to the console:

```
LOCAL aElements := { 1, 2, 3, 4, 5 }
AEVAL(aElements, { |element| QOUT(element) })
```

Files: Library is CLIPPER.LIB.

See Also: ?|?? @...SAY SET ALTERNATE SET CONSOLE SET PRINTER

RAT()
Return the position of the last occurrence of a substring

Syntax

RAT(<cSearch>, <cTarget>) --> nPosition

Arguments

<cSearch> is the character string to locate.

<cTarget> is the character string to search.

Returns

RAT() returns the position of <cSearch> within <cTarget> as an integer numeric value. If <cSearch> is not found, RAT() returns zero.

Description

RAT() is a character function that returns the position of the last occurrence of a character substring within another character string. It does this by searching the target string from the right. RAT() is like the AT() function which returns the position of the first occurrence of a substring within another string. RAT() is also like the $ operator which determines whether a substring is contained within a string.

Both the RAT() and AT() functions are used with SUBSTR(), LEFT(), and RIGHT() to extract substrings.

Examples

. This example uses RAT() to create a user-defined function, FilePath(), that extracts the path from a file specification. If the path is unspecified, FilePath() returns a null string (""):

```
? FilePath("C:\DBF\Sales.dbf")      // Result: C:\DBF\

FUNCTION FilePath( cFile )
   LOCAL nPos, cFilePath
   IF (nPos := RAT("\", cFile)) != 0
      cFilePath = SUBSTR(cFile, 1, nPos)
   ELSE
      cFilePath = ""
   ENDIF
   RETURN cFilePath
```

Files: Library is EXTEND.LIB.

See Also: AT() LEFT() RIGHT() STRTRAN() SUBSTR()

READEXIT()
Toggle Uparrow and Dnarrow as READ exit keys

Syntax

READEXIT([<lToggle>]) --> lCurrentState

Arguments

<lToggle> toggles the use of Uparrow and Dnarrow as READ exit keys. Specifying true (.T.) enables them as exit keys, and false (.F.) disables them.

Returns

READEXIT() returns the current setting as a logical value.

Description

READEXIT() is an environment function that reports the current state of Uparrow and Dnarrow as keys the user can press to exit a READ from the first or last Get object in a GetList. If the optional <lToggle> argument is specified, Uparrow and Dnarrow are either enabled or disabled as READ exit keys. At program startup, Uparrow and Dnarrow are not enabled as READ exit keys. Normally, READ exit keys include only PgUp, PgDn, Esc, or Return from the last GET.

Examples

. This example shows READEXIT() enabling Uparrow and Dnarrow exit keys before a READ then resetting them after the READ terminates:

```
cMyvar = SPACE(10)
lLastExit = READEXIT(.T.)   // Result: Turn on exit keys
//
@ 10, 10 SAY "Enter: " GET cMyvar
READ
READEXIT(lLastExit)         // Result: Restore previous setting
```

Files: Library is CLIPPER.LIB.

See Also: @...GET READ READINSERT()

READINSERT()
Toggle the current insert mode for READ and MEMOEDIT()

Syntax

READINSERT([<lToggle>]) --> lCurrentMode

Arguments

<lToggle> toggles the insert mode on or off. True (.T.) turns insert on, while false (.F.) turns insert off. The default is false (.F.) or the last user-selected mode in READ or MEMOEDIT().

Returns

READINSERT() returns the current insert mode state as a logical value.

Description

READINSERT() is an environment function that reports the current state of the insert mode for READ and MEMOEDIT(), and optionally sets the insert mode on or off depending on the value of <lToggle>. When READINSERT() returns false (.F.), and the user enters characters into a Get object's buffer during a READ or a MEMOEDIT(), characters are overwritten. When READINSERT() returns true (.T.), entered characters are inserted instead. The insert mode is a global setting belonging to the system and not any specific object.

You can execute READINSERT() prior to or during a READ or MEMOEDIT(). If used with READ, READINSERT() can be invoked within a WHEN or VALID clause of @...GET or within a SET KEY procedure. If used with MEMOEDIT(), it can be invoked with the user function as well as a SET KEY procedure.

Examples

. This example sets the insert mode prior to entering MEMOEDIT() and resets the mode when MEMOEDIT() terminates:

```
USE Sales NEW

// Turn on insert mode
lInsMode = READINSERT(.T.)
Sales->Notes := MEMOEDIT(Sales->Notes)
//
// Restore previous insert mode
READINSERT(lInsMode)
```

Files: Library is CLIPPER.LIB.

See Also: MEMOEDIT() READ READEXIT()

READKEY()*
Determine what key terminated a READ

Syntax

READKEY() --> nReadkeyCode

Returns

READKEY() returns a code representing the key pressed to exit a READ. In CA-Clipper, the following keys are the standard READ exit keys and their READKEY() return codes:

READKEY() Return Codes

Exit Key	Return Code
Uparrow	5
Dnarrow	2
PgUp	6
PgDn	7
Ctrl-PgUp	31
Ctrl-PgDn	30
Esc	12
Ctrl-End, Ctrl-W	14
Type past end	15
Return	15

Description

READKEY() is a keyboard function that emulates the READKEY() function in dBASE III PLUS. Its purpose is to determine what key the user pressed to terminate a READ. If UPDATED() is true (.T.), READKEY() returns the code plus 256. Uparrow and Dnarrow exit a READ only if READEXIT() returns true (.T.). The default value is false (.F.). To provide complete compatibility for these keys, execute a READEXIT (.T.) at the beginning of your main procedure.

READKEY() is supplied as a compatibility function and therefore its use is strongly discouraged. It is superseded entirely by LASTKEY() which determines the last keystroke fetched from the keyboard buffer. If the keystroke was a READ exit key, LASTKEY() will return the INKEY() code for that key. To determine whether any Get object's buffer was modified during a READ, it is superseded by the UPDATED() function.

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\READKEY.PRG.

See Also: @...GET LASTKEY() READ READEXIT() UPDATED() Inkey codes

READMODAL()
Activate a full-screen editing mode for a GetList

Syntax

READMODAL(<aGetList>) --> NIL

Arguments

<aGetList> is an array containing a list of Get objects to edit.

Returns

READMODAL() always returns NIL.

Description

READMODAL() is a user interface function that implements the full-screen editing mode for GETs, and is part of the open architecture Get system of CA-Clipper. READMODAL() is like the READ command, but takes a GetList array as an argument and does not reinitialize the GetList array when it terminates. Because of this, you can maintain multiple lists of Get objects and activate them any time in a program's execution as long as the array to activate is visible.

In order to retain compatibility with previous versions of CA-Clipper, the GET system in CA-Clipper is implemented using a public array called GetList. Each time an @...GET command executes, it creates a Get object and adds to the currently visible GetList array. The standard READ command is preprocessed into a call to READMODAL() using the GetList array as its argument. If the SAVE clause is not specified, the variable GetList is assigned an empty array after the READMODAL() function terminates.

Some of the functions in the Getsys.prg have been made public so that they can be used when implementing customized Get readers. These functions are listed in the table below.

Get System functions

Function	Description
GETACTIVE()	Return the currently active Get object
GETAPPLYKEY()	Apply a key to a Get object from within a Get reader
GETDOSETKEY()	Process SET KEY during Get editing
GETPOSTVALIDATE()	Postvalidate the current Get object
GETPREVALIDATE()	Prevalidate a Get object
GETREADER()	Execute standard READ behavior for a Get object
GETPREVALIDATE()	Prevalidate a Get object
READFORMAT()	Return, and optionally set the code block that implements a format (.fmt) file
READKILL()	Return, and optionally set whether the current Read should be exited
READUPDATED()	Return, and optionally set whether a Get has changed during a Read

For reference information on the Get objects and functions listed above, refer to the Get System chapter in the Programming and Utilities guide.

For more information on the supported keys in the default READMODAL() function, refer to the READ command reference in this chapter.

Files: Library is CLIPPER.LIB, source file is SOURCE\SYS\GETSYS.PRG.

See Also: @...GET READ

READVAR()
Return the current GET/MENU variable name

Syntax

READVAR() --> cVarName

Returns

READVAR() returns the name of the variable associated with the current Get object or the variable being assigned by the current MENU TO command as an uppercase character string.

Description

READVAR() is an environment function that, primarily, implements context-sensitive help for Get objects and lightbar menus. READVAR() only works during a READ or MENU TO command. If used during any other wait states such as ACCEPT, INPUT, WAIT, ACHOICE(), DBEDIT(), or MEMOEDIT(), it returns a null string (""). Access it within a SET KEY procedure, or within a user-defined function invoked from a WHEN or VALID clause of a Get object.

Examples

. This example implements a simple help system for Get objects using a database file to store the help text. When the user presses F1, the help database file is searched using READVAR() as the key value. If there is help text available, it is displayed in a window:

```
#include "Inkey.ch"
//
SET KEY K_F1 TO HelpLookup
cString = SPACE(10)
@ 5, 5 SAY "Enter:" GET cString
READ
RETURN

FUNCTION HelpLookup
   USE Help INDEX Help NEW
   SEEK READVAR()
   IF FOUND()
      DisplayHelp(Help->Topic)
   ELSE
      DisplayHelp("No help for " + READVAR())
   ENDIF
   CLOSE Help
   RETURN NIL

FUNCTION DisplayHelp( cTopic )
   LOCAL cScreen := SAVESCREEN(5,5,15,70),;
         cColor := SETCOLOR("BG+/B")
   //
   SET CURSOR OFF
   @ 5, 5 CLEAR TO 15, 70
   @ 5, 5 TO 15, 70 DOUBLE
   @ 5, 30 SAY " Help for " + READVAR() + " "
   MEMOEDIT(cTopic, 6, 7, 14, 68, .F.)
   //
   RESTSCREEN(5, 5, 15, 70, cScreen)
   SETCOLOR(cColor)
   SET CURSOR ON
   //
   RETURN NIL
```

Files: Library is CLIPPER.LIB.

See Also: @...GET MENU TO READ SET KEY

RECCOUNT()*
Determine the number of records in the current (.dbf) file

Syntax

RECCOUNT()* | LASTREC() --> nRecords

Returns

RECCOUNT() returns the number of physical records in the current database file as an integer numeric value. Filtering commands such as SET FILTER or SET DELETED have no effect on the return value. RECCOUNT() returns zero if there is no database file open in the current work area.

Description

RECCOUNT() is a database function that is a synonym for LASTREC(). By default, RECCOUNT() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

Examples

. This example illustrates the relationship between COUNT and RECCOUNT():

```
USE Sales NEW
? RECCOUNT()                      // Result: 84
//
SET FILTER TO Salesman = "1001"
COUNT TO nRecords
? nRecords                        // Result: 14
? RECCOUNT()                      // Result: 84
```

. This example uses an aliased expression to access the number of records in an unselected work area:

```
USE Sales NEW
USE Customer NEW
? RECCOUNT(), Sales->(RECCOUNT())
```

Files: Library is CLIPPER.LIB.

See Also: COUNT EOF() LASTREC()

RECNO()
Return the current record number of a work area

Syntax

RECNO() --> nRecord

Returns

RECNO() returns the current record number as an integer numeric value. If the work area contains a database file with zero records, RECNO() returns one, BOF() and EOF() both return true (.T.), and LASTREC() returns zero.

If the record pointer is moved past the last record, RECNO() returns LASTREC() + 1 and EOF() returns true (.T.). If an attempt is made to move before the first record, RECNO() returns the record number of the first logical record in the database file and BOF() returns true (.T.). If no database file is open, RECNO() will return a zero.

Description

RECNO() is a database function that returns the current record number in a work area. In the CA-Clipper database file scheme, each database file is physically ordered by record number. Each work area in turn maintains a pointer to the current record in its open database file. That record number is reported by RECNO(). The record numbering facility allows direct access to a record without sequentially scanning the database file to reach the specified record position.

RECNO(), typically, generalizes routines that process records by record number. This includes SET RELATION...TO RECNO() which links database files by record number. GO RECNO() also refreshes the current record data from disk.

By default, RECNO() operates on the currently selected work area. It operates on an unselected work area if you specify it as part of an aliased expression (see example below).

Examples

. This example queries RECNO() after deliberately moving the record pointer:

```
USE Customers NEW
GO 3
? RECNO()                       // Result: 3
GO TOP
? RECNO()                       // Result: 1
nRecord := 3
GO nRecord
? RECNO()                       // Result: 3
GO BOTTOM
SKIP
? RECNO(), LASTREC()            // Result: 11 10
```

. This example uses aliased expressions to query the value of RECNO() in unselected work areas:

```
USE Sales NEW
USE Customer NEW
//
? Sales->(RECNO())
? Customer->(RECNO())
```

Files: Library is CLIPPER.LIB.

See Also: BOF() EOF() GO LASTREC() SKIP

RECSIZE()
Determine the record length of a database (.dbf) file

Syntax

RECSIZE() --> nBytes

Returns

RECSIZE() returns, as a numeric value, the record length, in bytes, of the database file open in the current work area. RECSIZE() returns zero if no database file is open.

Description

RECSIZE() is a database function that determines the length of a record by summing the lengths of each field then adding one for the DELETED() status flag. When this value is multiplied by LASTREC(), the product is the amount of space occupied by the file's records.

RECSIZE() is useful in programs that perform automatic file backup. When used in conjunction with DISKSPACE(), the RECSIZE() function can assist in ensuring that sufficient free space exists on the disk before a file is stored.

By default, RECSIZE() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below).

Examples

. The following user-defined function, DbfSize(), uses RECSIZE() to calculate the size of the current database file:

```
FUNCTION DbfSize
   RETURN ((RECSIZE() * LASTREC()) + HEADER() + 1)
```

. This example illustrates the use of RECSIZE() to determine the record length of database files open in unselected work areas:

```
USE Customer NEW
USE Sales NEW
//
? RECSIZE(), Customer->(RECSIZE())
? DbfSize(), Customer->(DbfSize())
```

Files: Library is EXTEND.LIB.

See Also: DISKSPACE() FIELDNAME() HEADER() LASTREC()

REPLICATE()
Return a string repeated a specified number of times

Syntax

REPLICATE(<cString>, <nCount>) --> cRepeatedString

Arguments

<cString> is the character string to repeat.

<nCount> is the number of times to repeat <cString>.

Returns

REPLICATE() returns a character string up to a maximum of 65,535 (64K) bytes in length. Specifying a zero as the <nCount> argument returns a null string ("").

Description

REPLICATE() is a character function that repeatedly displays, prints, or stuffs the keyboard with one or more characters. REPLICATE() is like the SPACE() function which returns a specified number of space characters.

Examples

. These examples demonstrate REPLICATE() repeating strings:

```
? REPLICATE("*", 5)           // Result: *****
? REPLICATE("Hi ", 2)         // Result: Hi Hi
? REPLICATE(CHR(42), 5)       // Result: *****
```

. This example uses REPLICATE() to stuff the keyboard with several Dnarrow keys:

```
#include "Inkey.ch"
KEYBOARD REPLICATE(CHR(K_DN), 25)
```

Files: Library is CLIPPER.LIB.

See Also: SPACE()

RESTSCREEN()
Display a saved screen region to a specified location

Syntax

RESTSCREEN([<nTop>], [<nLeft>],
[<nBottom>], [<nRight>], <cScreen>) --> NIL

Arguments

<nTop>, **<nLeft>**, **<nBottom>**, and **<nRight>** define the coordinates of the screen information contained in <cScreen>. If the <cScreen> was saved without coordinates to preserve the entire screen, no screen coordinates are necessary with RESTSCREEN().

<cScreen> is a character string containing the saved screen region.

Returns

RESTSCREEN() always returns NIL.

Description

RESTSCREEN() is a screen function that redisplays a screen region saved with SAVESCREEN(). The target screen location may be the same as or different than the original location when the screen region was saved. If you specify a new screen location, the new screen region must be the same size or you will get ambiguous results. To use RESTSCREEN() to restore screen regions saved with SAVE SCREEN, specify the region coordinates as 0, 0, MAXROW(), MAXCOL().

Warning! SAVE SCREEN, RESTORE SCREEN, SAVESCREEN(), and RESTSCREEN() are supported when using the default (IBM PC memory mapped) screen driver. Other screen drivers may not support saving and restoring screens.

Examples

. This example demonstrates RESTSCREEN() as part of a general purpose popup menu function, PopMenu():

```
? PopMenu({1, 1, 3, 10, {"ItemOne", "ItemTwo"}, ;
      "BG+/B"})

FUNCTION PopMenu( aList )
   LOCAL cScreen, nChoice, cOldColor := ;
            SETCOLOR(aList[6])
   cScreen := SAVESCREEN(aList[1], aList[2],;
       aList[3], aList[4])
   @ aList[1], aList[2], TO aList[3], aList[4] DOUBLE
   nChoice := ACHOICE(++aList[1], ++aList[2],;
      --aList[3], --aList[4], aList[5])
   SETCOLOR(cOldColor)
   RESTSCREEN(--aList[1], --aList[2], ++aList[3],;
       ++aList[4], cScreen)
   RETURN nChoice
```

Files: Library is EXTEND.LIB.

See Also: SAVESCREEN()

RIGHT()
Return a substring beginning with the rightmost character

Syntax

RIGHT(<cString>, <nCount>) --> cSubString

Arguments

<cString> is the character string from which to extract characters.

<nCount> is the number of characters to extract.

Returns

RIGHT() returns the rightmost <nCount> characters of <cString>. If <nCount> is zero, RIGHT() returns a null string (""). If <nCount> is negative or larger than the length of the character string, RIGHT() returns <cString>. The maximum string size is 65,535 (64K) bytes.

Description

RIGHT() is a character function that extracts a substring beginning with the rightmost character in <cString>. It is the same as the character expression, SUBSTR(<cString>, -<nCount>). For example, RIGHT("ABC", 1) is the same as SUBSTR("ABC", -1). RIGHT() is related to LEFT() which extracts a substring beginning with the leftmost character in <cString>.

The RIGHT(), LEFT(), and SUBSTR() functions are often used with both the AT() and RAT() functions to locate either the first and/or the last position of a substring before extracting it.

Examples

. This example shows the relationship between RIGHT() and SUBSTR():

```
? RIGHT("ABCDEF", 3)                    // Result: DEF
? SUBSTR("ABCDEF", -3)                  // Result: DEF
```

. This example extracts a substring from the end of another string up to the last occurrence of a comma:

```
LOCAL cName := "James,William"
? RIGHT(cName,;
LEN(cName) - RAT(",", cName) - 1)       // Result: William
```

Files: Library is EXTEND.LIB.

See Also: LEFT() LTRIM() RTRIM() STUFF() SUBSTR()

RLOCK()
Lock the current record in the active work area

Syntax

RLOCK() --> lSuccess

Returns

RLOCK() returns true (.T.) if the record lock is obtained; otherwise, it returns false (.F.).

Description

RLOCK() is a network function that locks the current record, preventing other users from updating the record until the lock is released. RLOCK() provides a shared lock, allowing other users read-only access to the locked record while allowing only the current user to modify it. A record lock remains until another record is locked, an UNLOCK is executed, the current database file is closed, or an FLOCK() is obtained on the current database file.

For each invocation of RLOCK(), there is one attempt to lock the current record, and the result is returned as a logical value. An attempt to obtain a record lock fails if another user currently has a file or record lock, or EXCLUSIVE USE of the database file. An attempt to RLOCK() in an empty database returns true (.T.).

By default, RLOCK() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression (see example below). This feature is useful since RLOCK() does not automatically attempt a record lock for related files.

As a general rule, RLOCK() operates solely on the current record. This includes the following commands:

- @...GET
- DELETE (single record)
- RECALL (single record)
- REPLACE (single record)

Refer to the Network Programming chapter in the Programming and Utilities guide for more information.

Notes

- **SET RELATION:** CA-Clipper does not automatically lock all records in the relation chain when you lock the current work area record. Also, an UNLOCK has no effect on related work areas.

Examples

- This example deletes a record in a network environment, using RLOCK():

```
USE Customer INDEX CustName SHARED NEW
SEEK "Smith"
IF FOUND()
   IF RLOCK()
      DELETE
      ? "Smith deleted"
   ELSE
      ? "Record in use by another"
   ENDIF
```

```
ELSE
   ? "Smith not in Customer file"
ENDIF
CLOSE
```

. This example specifies RLOCK() as an aliased expression to lock a record in an unselected work area:

```
USE Sales SHARED NEW
USE Customer SHARED NEW
//
IF !Sales->(RLOCK())
   ? "The current Sales record is in use by another"
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: APPEND BLANK FLOCK() SET EXCLUSIVE* UNLOCK USE

ROUND()
Return a numeric value rounded to a specified number of digits

---

Syntax

ROUND(<nNumber>, <nDecimals>) --> nRounded

Arguments

<nNumber> is the numeric value to round.

<nDecimals> defines the number of decimal places to retain.
Specifying a negative <nDecimals> value rounds whole number digits.

Returns

ROUND() returns a numeric value.

Description

ROUND() is a numeric function that rounds <nNumber> to the number of places specified by <nDecimals>. Specifying a zero or negative value for <nDecimals> allows rounding of whole numbers. A negative <nDecimals> indicates the number of digits to the left of the decimal point to round. Digits between five to nine, inclusive, are rounded up. Digits below five are rounded down.

The display of the return value does not obey the DECIMALS setting unless SET FIXED is ON. With SET FIXED OFF, the display of the return value contains as many decimal digits as you specify for <nDecimals>, or zero, if <nDecimals> is less than one.

Examples

. These examples round values with decimal digits:

```
SET DECIMALS TO 2
SET FIXED ON
//
? ROUND(10.4, 0)                    // Result: 10.00
? ROUND(10.5, 0)                    // Result: 11.00
? ROUND(10.51, 0)                   // Result: 11.00
? ROUND(10.49999999999999, 2)       // Result: 10.50
```

. These examples use a negative <nDecimals> argument to round numeric values to whole number values:

```
? ROUND(101.99, -1)                 // Result: 100.00
? ROUND(109.99, -1)                 // Result: 110.00
? ROUND(109.99, -2)                 // Result: 100.00
```

Files: Library is CLIPPER.LIB.

See Also: INT() SET DECIMALS SET FIXED STR() VAL()

ROW()
Return the screen row position of the cursor

Syntax

ROW() --> nRow

Returns

ROW() returns the cursor row position as an integer numeric value. The range of the return value is zero to MAXROW().

Description

ROW() is a screen function that returns the current row or line position of the screen cursor. The value of ROW() is updated by both console and full-screen commands and functions. @...SAY only updates ROW() when the current DEVICE is the SCREEN.

ROW() is used with COL() and all variations of the @ command to position the cursor to a new line relative to the current line. In particular, you can use ROW() and COL() to create screen position-independent procedures or functions where you pass the upper left row and column as parameters.

ROW() is related to PROW() and PCOL() which track the current printhead position instead of the screen cursor position.

Examples

. In this example, ROW() simulates the LIST command, displaying text on the same line but in different columns:

```
LOCAL nRow
USE Customer INDEX CustName NEW
DO WHILE .NOT. EOF()
   CLS
   @ 1, 1 SAY PADR("Name", LEN(CustName))
   @ ROW(), COL() + 2 SAY PADR("Address", ;
         LEN(Address))
   @ ROW(), COL() + 2 SAY PADR("Phone", LEN(Phone))
   nRow = 0
   DO WHILE nRow++ <= 15 .AND. (!EOF())
      @ ROW() + 1, 1 SAY CustName
      @ ROW(), COL() + 2 SAY Address
      @ ROW(), COL() + 2 SAY Phone
      SKIP
   ENDDO
   WAIT
ENDDO
CLOSE Customer
```

Files: Library is CLIPPER.LIB.

See Also: ?|?? @...SAY COL() MAXROW() PROW() SET DEVICE

RTRIM()
Remove trailing spaces from a character string

Syntax

[R]TRIM(<cString>) --> cTrimString

Arguments

<cString> is the character string to copy without trailing spaces.

Returns

RTRIM() returns a copy of <cString> with the trailing spaces removed. If <cString> is a null string ("") or all spaces, RTRIM() returns a null string ("").

Description

RTRIM() is a character function that formats character strings. It is useful when you want to delete trailing spaces while concatenating strings. This is typically the case with database fields which are stored in fixed-width format. For example, you can use RTRIM() to concatenate first and last name fields to form a name string.

RTRIM() is related to LTRIM() which removes leading spaces, and ALLTRIM() which removes both leading and trailing spaces. The inverse of ALLTRIM(), LTRIM(), and RTRIM() are the PADC(), PADR(), and PADL() functions which center, right-justify, or left-justify character strings by padding them with fill characters.

Examples

. This is a user-defined function in which RTRIM() formats city, state, and zip code fields for labels or form letters:

```
FUNCTION CityState(cCity, cState, cZip)
   RETURN RTRIM(cCity) + ", " ;
    + RTRIM(cState) + "  " + cZip
```

. In this example the user-defined function, CityState(), displays a record from Customer.dbf:

```
USE Customer INDEX CustName NEW
SEEK "Kate"
? CityState(City, State, ZipCode)
// Result: Athens, GA 10066
```

Files: Library is CLIPPER.LIB.

See Also: ALLTRIM() LTRIM() PAD() SUBSTR()

SAVESCREEN()
Save a screen region for later display

Syntax

SAVESCREEN([<nTop>], [<nLeft>],
[<nBottom>], [<nRight>]) --> cScreen

Arguments

<nTop>, **<nLeft>**, **<nBottom>**, and **<nRight>** define the coordinates of the screen region to save. If either <nBottom> or <nRight> is greater than MAXROW() or MAXCOL(), the screen is clipped. If you specify no coordinates, the entire screen (i.e., from 0,0 to MAXROW(), MAXCOL()) is saved.

Returns

SAVESCREEN() returns the specified screen region as a character string.

Description

SAVESCREEN() is a screen function that saves a screen region to a variable of any storage class including a field variable. Later, you can redisplay the saved screen image to the same or a new location using RESTSCREEN(). Screen regions are usually saved and restored when using a popup menu routine or dragging a screen object.

Warning! SAVE SCREEN, RESTORE SCREEN, SAVESCREEN(), and RESTSCREEN() are supported when using the default (IBM PC memory mapped) screen driver. Other screen drivers may not support saving and restoring screens.

Examples

. The following user-defined function creates a popup menu using ACHOICE() with SAVESCREEN() and RESTSCREEN(), returning the selection in the array of choices:

```
FUNCTION PopMenu( nTop, nLeft, nBottom, nRight, ;
                  aItems, cColor )
   LOCAL cScreen, nChoice, cLastColor := ;
         SETCOLOR(cColor)
   //
   cScreen:= SAVESCREEN(nTop, nLeft, nBottom, nRight)
   @ nTop, nLeft TO nBottom, nRight DOUBLE
   //
   nChoice:= ACHOICE(++nTop, ++nLeft, ;
               --nBottom, --nRight, aItems)
   //
   RESTSCREEN(--nTop, --nLeft, ++nBottom, ++nRight, ;
                cScreen)
   SETCOLOR(cLastColor)
   RETURN nChoice
```

Files: Library is EXTEND.LIB.

See Also: RESTSCREEN()

SCROLL()
Scroll a screen region up or down, right or left

Syntax

```
SCROLL([<nTop>], [<nLeft>],
   [<nBottom>], [<nRight>], [<nVert>] [<nHoriz>])
   --> NIL
```

Arguments

<nTop>, **<nLeft>**, **<nBottom>**, and **<nRight>** define the scroll region coordinates. Row and columns values can range from 0, 0 to MAXROW(), MAXCOL(). If you do not specify coordinate arguments, the dimensions of the visible display are used.

<nVert> defines the number of rows to scroll, vertically. A positive value scrolls up the specified number of rows. A negative value scrolls down the specified number of rows. A value of zero disables vertical scrolling. If <nVert> is not specified, zero is assumed.

<nHoriz> defines the number of rows to scroll horizontally. A positive value scrolls left the specified number of columns. A negative value scrolls right the specified number of columns. A value of zero disables horizontal scrolling. If <nHoriz> is not specified, zero is assumed.

If you supply neither <nVert> or <nHoriz> parameters to SCROLL(), the area specified by the first four parameters will be blanked.

Warning! Horizontal scrolling is not supported on all of the alternate terminal drivers (i.e. ANSITERM, NOVTERM, PCBIOS).

Returns

SCROLL() always returns NIL.

Description

SCROLL() is a screen function that scrolls a screen region up or down a specified number of rows. When a screen scrolls up, the first line of the region is erased, all other lines are moved up, and a blank line is displayed in the current standard color on the bottom line of the specified region. If the region scrolls down, the operation is reversed. If the screen region is scrolled more than one line, this process is repeated.

SCROLL() is used primarily to display status information into a defined screen region. Each time a new message is displayed, the screen region scrolls up one line and a new line displays at the bottom.

Examples

. This user-defined function displays a message string at the bottom of a screen region after scrolling the region up one line:

```
FUNCTION ScrollUp( nTop, nLeft, nBottom, nRight, ;
                   expDisplay )
   //
   SCROLL(nTop, nLeft, nBottom, nRight, 1)
   @ nBottom, nLeft SAY expDisplay
   //
   RETURN NIL
```

Files: Library is CLIPPER.LIB.

See Also: @...BOX @...CLEAR @...TO CLEAR SCREEN

SECONDS()
Return the number of seconds elapsed since midnight

Syntax

SECONDS() --> nSeconds

Returns

SECONDS() returns the system time as a numeric value in the form seconds.hundredths. The numeric value returned is the number of seconds elapsed since midnight, and is based on a twenty-four hour clock in a range from zero to 86399.

Description

SECONDS() is a time function that provides a simple method of calculating elapsed time during program execution, based on the system clock. It is related to the TIME() function which returns the system time as a string in the form hh:mm:ss.

Examples

. This example contrasts the value of TIME() with SECONDS():

```
? TIME()               // Result: 10:00:00
? SECONDS()            // Result: 36000.00
```

. This example uses SECONDS() to track elapsed time in seconds:

```
LOCAL nStart, nElapsed
nStart:= SECONDS()
.
. <statements>
.
nElapsed:= SECONDS() - nStart
? "Elapsed: " + LTRIM(STR(nElapsed)) + " seconds"
```

Files: Library is CLIPPER.LIB.

See Also: TIME()

SELECT()
Determine the work area number of a specified alias

Syntax

SELECT([<cAlias>]) --> nWorkArea

Arguments

<cAlias> is the target work area alias name.

Returns

SELECT() returns the work area of the specified alias as a integer numeric value.

Description

SELECT() is a database function that determines the work area number of an alias. The number returned can range from zero to 250. If <cAlias> is not specified, the current work area number is returned. If <cAlias> is specified and the alias does not exist, SELECT() returns zero.

Note: The SELECT() function and SELECT command specified with an extended expression argument look somewhat alike. This shouldn't be a problem since the SELECT() function is not very useful on a line by itself

Examples

. This example uses SELECT() to determine which work area USE...NEW selected:

```
USE Sales NEW
SELECT 1
? SELECT("Sales")            // Result: 4
```

. To reselect the value returned from the SELECT() function, use the SELECT command with the syntax, SELECT (<idMemvar>), like this:

```
USE Sales NEW
nWorkArea:= SELECT()
USE Customer NEW
SELECT (nWorkArea)
```

Files: Library is CLIPPER.LIB.

See Also: ALIAS() SELECT USE USED()

SET()
Inspect or change a system setting

Syntax

SET(<nSpecifier>, [<expNewSetting>], [<lOpenMode>])
--> CurrentSetting

Arguments

<nSpecifier> is a numeric value that identifies the setting to be inspected or changed. <nSpecifier> should be supplied as a manifest constant (see below).

<expNewSetting> is an optional argument that specifies a new value for the <nSpecifier>. The type of <expNewSetting> depends on <nSpecifier>.

<lOpenMode> is a logical value that indicates whether or not files are opened for the following settings:

_SET_ALTFILE, _SET_PRINTFILE, _SET_EXTRAFILE

should be truncated or opened in append mode. A value of false (.F.) means the file should be truncated. A value of true (.T.) means the file should be opened in append mode. In either case, if the file does not exist, it is created.

If this argument is not specified, the default is append mode.

Returns

SET() returns the current value of the specified setting.

Description

SET() is a system function that lets you inspect or change the values of the CA-Clipper system settings. For information on the meaning and legal values for a particular setting, refer to the associated command or function.

Use a manifest constant to specify the setting to be inspected or changed. These constants are defined in a header file called Set.ch. This header file should be included at the top of any source file which uses SET().

Set.ch also defines a constant called _SET_COUNT. This constant is equal to the number of settings that can be changed or inspected with SET(), allowing the construction of a generic function that preserves all settings (see example below).

Note: The numeric values of the manifest constants in Set.ch are version-dependent and should never be used directly; the manifest constants should always be used.

If <nSpecifier> or <expNewSetting> is invalid, the call to SET() is ignored.

Set Values Defined in Set.ch

Constant	Value Type	Associated Command or Function
_SET_EXACT	Logical	SET EXACT
_SET_FIXED	Logical	SET FIXED
_SET_DECIMALS	Numeric	SET DECIMALS
_SET_DATEFORMAT	Character	SET DATE
_SET_EPOCH	Numeric	SET EPOCH

```
_SET_PATH           Character      SET PATH
_SET_DEFAULT        Character      SET DEFAULT
_SET_EXCLUSIVE      Logical        SET EXCLUSIVE
_SET_SOFTSEEK       Logical        SET SOFTSEEK
_SET_UNIQUE         Logical        SET UNIQUE
_SET_DELETED        Logical        SET DELETED
_SET_CANCEL         Logical        SETCANCEL()
_SET_DEBUG          Logical        ALTD()
_SET_COLOR          Character      SETCOLOR()
_SET_CURSOR         Numeric        SETCURSOR()
_SET_CONSOLE        Logical        SET CONSOLE
_SET_ALTERNATE      Logical        SET ALTERNATE
_SET_ALTFILE        Character      SET ALTERNATE TO
_SET_DEVICE         Character      SET DEVICE
_SET_PRINTER        Logical        SET PRINTER
_SET_PRINTFILE      Character      SET PRINTER TO
_SET_MARGIN         Numeric        SET MARGIN
_SET_BELL           Logical        SET BELL
_SET_CONFIRM        Logical        SET CONFIRM
_SET_ESCAPE         Logical        SET ESCAPE
_SET_INSERT         Logical        READINSERT()
_SET_EXIT           Logical        READEXIT()
_SET_INTENSITY      Logical        SET INTENSITY
_SET_SCOREBOARD     Logical        SET SCOREBOARD
_SET_DELIMITERS     Logical        SET DELIMITERS
_SET_DELIMCHARS     Character      SET DELIMITERS TO
_SET_WRAP           Logical        SET WRAP
_SET_MESSAGE        Numeric        SET MESSAGE
_SET_MCENTER        Logical        SET MESSAGE
```

Note: _SET_EXTRAFILE and _SET_SCROLLBREAK have no corresponding commands. _SET_EXTRAFILE lets you specify an additional alternate file, and _SET_SCROLLBREAK lets you toggle the interpretation of Ctrl-S.

Examples

. In this example a user-defined function preserves or restores all global settings. This function might be used on entry to a subsystem to ensure the subsystem does not affect the state of the program that called it:

```
#include "Set.ch"
//

FUNCTION SetAll( aNewSets )
   LOCAL aCurrentSets[_SET_COUNT], nCurrent
   IF ( aNewSets != NIL )   // Set new and return current
      FOR nCurrent := 1 TO _SET_COUNT
         aCurrentSets[nCurrent] := ;
            SET(nCurrent, aNewSets[nCurrent])
      NEXT
   ELSE                     // Just return current
      FOR nCurrent := 1 TO _SET_COUNT
         aCurrentSets[nCurrent] := SET(nCurrent)
      NEXT
   ENDIF
   RETURN (aCurrentSets)
```

Files: Library is CLIPPER.LIB, header file is Set.ch.

SETBLINK()
Toggle asterisk (*) interpretation in SETCOLOR() string between blinking and background intensity

---

Syntax

SETBLINK([<lToggle>]) --> lCurrentSetting

Arguments

<lToggle> changes the meaning of the asterisk (*) character when it is encountered in a SETCOLOR() string. Specifying true (.T.) sets character blinking on and false (.F.) sets background intensity. The default is true (.T.).

Returns

SETBLINK() returns the current setting as a logical value.

Description

SETBLINK() is an environment function that toggles the blinking/background intensity attribute and reports the current state of SETBLINK(). When SETBLINK() is on, characters written to the screen can be made to blink by including an asterisk (*) in a color string passed to SETCOLOR(). When SETBLINK() is off, the asterisk (*) causes the background color to be intensified instead. Thus, blinking and background intensity attributes are not available at the same time.

Note: This function is meaningful only on the IBM PC or compatible computers with CGA, EGA, or VGA display hardware.

Examples

. This example saves the current SETBLINK() state before passing control to a user-defined function. Upon return, SETBLINK() is restored to its original value:

```
lOldBlink := SETBLINK()
MyFunc()
SETBLINK(lOldBlink)
```

Files: Library is CLIPPER.LIB.

See Also: SETCOLOR() SET COLOR*

SETCANCEL()
Toggle Alt-C and Ctrl-Break as program termination keys

Syntax

SETCANCEL([<lToggle>]) --> lCurrentSetting

Arguments

<lToggle> changes the availability of Alt-C and Ctrl-Break as termination keys. Specifying true (.T.) allows either of these keys to terminate an application and false (.F.) disables both keys. The default is true (.T.).

Returns

SETCANCEL() returns the current setting as a logical value.

Description

SETCANCEL() is a keyboard function that toggles the state of the termination keys, Alt-C and Ctrl-Break, and reports the current state of SETCANCEL(). Use SETCANCEL() when you want to suppress a user's ability to terminate a program without using the specified method.

Note that if Alt-C or Ctrl-Break is redefined with SET KEY, the SET KEY definition takes precedence even if SETCANCEL() returns true (.T.).

Warning! When SETCANCEL() has been set false (.F.), the user cannot terminate a runaway program unless you provide an alternative escape mechanism.

Examples

. This example provides an escape route from a wait state with SETCANCEL() set off:

```
#define K_ALTC   302
//
SETCANCEL(.F.)              // Disable termination keys
SET KEY K_ALTC TO AltC      // Redefine Alt-C
.
. <statements>
.
RETURN

FUNCTION AltC
   LOCAL cScreen, nChoice, cLastColor := ;
         SETCOLOR("W/B, N/G")
   //
   SAVE SCREEN TO cScreen
   @ 6, 20 CLEAR TO 9, 58
   @ 6, 20 TO 9, 58 DOUBLE
   @ 7, 26 SAY "Alt-C: Do you want to quit?"
   @ 8, 35 PROMPT " Yes "
   @ 8, 41 PROMPT " No "
   MENU TO nChoice
   SETCOLOR(cLastColor)
   RESTORE SCREEN FROM cScreen
   //
   IF nChoice = 1
      QUIT
   ENDIF
   //
   RETURN NIL
```

Files: Library is CLIPPER.LIB.

See Also: ALTD() SET ESCAPE SET KEY SETKEY()

SETCOLOR()
Return the current colors and optionally set new colors

Syntax

SETCOLOR([<cColorString>]) --> cColorString

Arguments

<cColorString> is a character string containing a list of color attribute settings for subsequent screen painting. The following is a list of settings and related scopes:

Color Settings

Setting	Scope
Standard	All screen output commands and functions
Enhanced	GETs and selection highlights
Border	Border around screen, not supported on EGA and VGA
Background	Not supported
Unselected	Unselected GETs

Each setting is a foreground and background color pair separated by the slash (/) character and followed by a comma. All settings are optional. If a setting is skipped, its previous value is retained with only new values set. Settings may be skipped within the list or left off the end as illustrated in the examples below.

Returns

SETCOLOR() returns the current color settings as a character string.

Description

SETCOLOR() is a screen function that saves the current color setting or sets new colors for subsequent screen painting. A color string is made from several color settings, each color corresponding to different regions of the screen. As stated above, each setting is made up of a foreground and background color. Foreground defines the color of characters displayed on the screen. Background defines the color displayed behind the character. Spaces and nondisplay characters display as background.

In CA-Clipper, the settings that define color behavior are:

Standard: The standard setting governs all console, full-screen, and interface commands and functions when displaying to the screen. This includes commands such as @...PROMPT, @...SAY, and ?, and functions such as ACHOICE(), DBEDIT(), and MEMOEDIT().

Enhanced: The enhanced setting governs highlighted displays. This includes GETs with INTENSITY ON, and the MENU TO, DBEDIT(), and ACHOICE() selection highlight.

Border: The border is an area around the screen that cannot be written to.

Background: The background is not supported.

Unselected: The unselected setting indicates input focus by displaying the current GET in the enhanced color while other GETs are displayed in the unselected color.

In addition to colors, foreground settings can have high intensity and/or blinking attributes. With a monochrome display, high intensity

enhances brightness of painted text. With a color display, high intensity changes the hue of the specified color. For example, "N" displays foreground text as black where "N+" displays the same text as gray. High intensity is denoted by "+". The blinking attribute causes the foreground text to flash on and off at rapid intervals. Blinking is denoted with "*". The attribute character can occur anywhere in the setting string, but is always applied to the foreground color regardless of where it occurs.

The following colors are supported:

List of Colors

Color	Letter	Monochrome
Black	N, Space	Black
Blue	B	Underline
Green	G	White
Cyan	BG	White
Red	R	White
Magenta	RB	White
Brown	GR	White
White	W	White
Gray	N+	White
Bright Blue	B+	Bright Underline
Bright Green	G+	Bright White
Bright Cyan	BG+	Bright White
Bright Red	R+	Bright White
Bright Magenta	RB+	Bright White
Yellow	GR+	Bright White
Bright White	W+	Bright White
Black	U	Underline
Inverse Video	I	Inverse Video
Blank	X	Blank

Notes

- **Arguments are not specified:** Unlike SET COLOR TO, SETCOLOR() with no argument does not restore colors to their default values.

- **Color numbers:** SETCOLOR() supports color letter combinations, but not color number combinations.

Examples

- This example assigns the current color setting to the variable, cColor:

```
cColor:= SETCOLOR()
```

- This example uses SETCOLOR() to save the current color setting and set a new one.

```
cNewColor:= "BR+/N, R+/N"
cOldColor:= SETCOLOR(cNewColor)
```

- This example uses SET COLOR TO to reset the default colors:

```
SET COLOR TO
? SETCOLOR()            // Result: W/N, N/W, N, N, N/W
```

- These two examples specify SETCOLOR() with missing settings:

```
// Settings left off the end
SETCOLOR("W/N, BG+/B")
//
// Settings skipped within the list
```

```
SETCOLOR("W/N, BG+/B,,,W/N")
```

. This example uses SETCOLOR() with ISCOLOR() to set the colors, depending on the screen type:

```
FUNCTION DefaultColors
   IF ISCOLOR()
      cForm := "W+/N, BG+/B,,,W/N"
      cDialog := "N/N+, BG+/B,,,N/N+"
      cAlert := "W+/R, BG+/B,,,W+/R"
   ELSE
      cForm := "W+/N, N/W,,,W/N"
      cDialog := "W+/N, N/W,,,W/N"
      cAlert := "W+/N, N/W,,,W/N"
   ENDIF
   RETURN NIL
```

Files: Library is CLIPPER.LIB.

See Also: ISCOLOR() SET COLOR* SET INTENSITY Colors

SETCURSOR()
Set the cursor shape

Syntax

SETCURSOR([<nCursorShape>]) --> nCurrentSetting

Arguments

<nCursorShape> is a number indicating the shape of the cursor. For simpler coding, the header file Setcurs.ch provides descriptive names for the various cursor shapes as shown in the table below:

Cursor Shapes

Shape	Value	Setcurs.ch
None	0	SC_NONE
Underline	1	SC_NORMAL
Lower half block	2	SC_INSERT
Full block	3	SC_SPECIAL1
Upper half block	4	SC_SPECIAL2

Returns

SETCURSOR() returns the current cursor shape as a numeric value.

Description

SETCURSOR() is an environment function that controls the shape of the screen cursor. The actual shape is dependent on the current screen driver. The specified shapes appear on IBM PC and compatible computers. On other computers, the appearance may differ for each value specified.

SETCURSOR(0) is the same as SET CURSOR OFF, and any positive integer value of <nCursorShape> less than 5 is the same as SET CURSOR ON. The cursor will display as the selected shape.

Examples

. This example uses SETCURSOR() to turn on a full block cursor for the subsequent READ. When the READ terminates SETCURSOR() turns off the cursor:

```
#include "Setcurs.ch"
//
USE Customer NEW
@ 10, 10 GET Customer->Name
@ 11, 10 GET Customer->Phone
//
SETCURSOR(SC_SPECIAL1)        // Change cursor to a block
READ
SETCURSOR(SC_NONE)            // Turn off cursor
```

Files: Library is CLIPPER.LIB, header file is Setcurs.ch.

See Also: SET CONSOLE SET CURSOR SETPOS()

SETKEY()
Assign an action block to a key

Syntax

SETKEY(<nInkeyCode>, [<bAction>]) --> bCurrentAction

Arguments

<nInkeyCode> is the INKEY() value of the key to be associated or queried.

<bAction> specifies a code block that is automatically executed whenever the specified key is pressed during a wait state.

Returns

SETKEY() returns the action block currently associated with the specified key, or NIL if the specified key is not currently associated with a block.

Description

SETKEY() is a keyboard function that sets or queries the automatic action associated with a particular key during a wait state. A wait state is any mode that extracts keys from the keyboard except for INKEY(), but including ACHOICE(), DBEDIT(), MEMOEDIT(), ACCEPT, INPUT, READ and WAIT. Up to 32 keys may be assigned at any one time. At startup, the system automatically assigns the F1 key to execute a procedure or user-defined function named Help.

When an assigned key is pressed during a wait state, the EVAL() function evaluates the associated <bAction> and the parameters, PROCNAME(), PROCLINE(), and READVAR(). It is, however, not necessary to list arguments when specifying <bAction> if you do not plan to use them within the action block.

SETKEY() is like the SET KEY command which associates a procedure invocation with a key.

Examples

. This code fragment associates an action block with a key, then, after getting a key using INKEY(), executes it with the EVAL() function:

```
#include "Inkey.ch"
SETKEY(K_DN, {|cProc, nLine, cVar| MyProc(cProc, ;
                  nLine, cVar)})
.
. <statements>
.
DO WHILE .T.
   nKey := INKEY(0)
   DO CASE
   CASE (bAction := SETKEY(nKey)) != NIL
      EVAL(bAction, PROCNAME(), PROCLINE(), READVAR())
   CASE nKey = K_PGUP
      Previous()
   CASE nKey = K_PGDN
      Next()
   CASE nKey = K_ESC
      EXIT
   ENDCASE
ENDDO
```

Files: Library is CLIPPER.LIB, header is Inkey.ch.

See Also: EVAL() INKEY() SET KEY Inkey codes

SETMODE()
Change display mode to a specified number of rows and columns

---

Syntax

SETMODE(<nRows>, <nCols>) --> lSuccess

Arguments

<nRows> is the number of rows in the desired display mode.

<nCols> is the number of columns in the desired display mode.

Returns

SETMODE() returns true (.T.) if the mode change was successful;
otherwise, it returns false (.F.).

Description

SETMODE() is an environment function that attempts to change the mode of the display hardware to match the number of rows and columns specified. The change in screen size is reflected in the values returned by MAXROW() and MAXCOL().

Examples

. This example switches to a 43-line display mode:

```
IF SETMODE(43, 80)
   ? "43-line mode successfully set"
ELSE
   ? "43-line mode not available"
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: MAXCOL() MAXROW()

SETPOS()
Move the cursor to a new position

Syntax

SETPOS(<nRow>, <nCol>) --> NIL

Arguments

<nRow> and **<nCol>** define the new screen position of the cursor.
These values may range from 0, 0 to MAXROW(), MAXCOL().

Returns

SETPOS() always returns NIL.

Description

SETPOS() is an environment function that moves the cursor to a new position on the screen. After the cursor is positioned, ROW() and COL() are updated accordingly. To control the shape and visibility of the cursor, use the SETCURSOR() function.

Examples

. This example moves the cursor to a new position then displays a string to the screen using a console command, ??:

```
SETPOS(1, 1)
?? "Hello world"
```

Files: Library is CLIPPER.LIB.

See Also: COL() ROW() SET CURSOR SETCURSOR()

SETPRC()
Set PROW() and PCOL() values

Syntax

SETPRC(<nRow>, <nCol>) --> NIL

Arguments

<nRow> is the new PROW() value.

<nCol> is the new PCOL() value.

Returns

SETPRC() always returns NIL.

Description

SETPRC() is a printer function that sends control codes to the printer without changing the tracking of the printhead position. When CA-Clipper prints, it updates the PCOL() value with the number of characters sent to the printer. There is no discrimination between printable or nonprintable characters. If, for example, a string of ten characters sent to the printer contains two characters interpreted by the printer as a control code, the CA-Clipper PCOL() value is incremented by ten, while the true printhead position is moved only by eight. This can lead to alignment problems. Using SETPRC(), you can compensate for control codes by resetting PCOL() as shown in the example below.

SETPRC() also suppresses page ejects when printing with @...SAY. This is important when the next row position is smaller than the current row and an EJECT has not been issued. In this situation, CA-Clipper issues an automatic page eject if the next row print position is less than the current PROW() value. Using SETPRC(), you can set PROW() to a number less than the current row, thus suppressing the automatic EJECT.

Examples

. This user-defined function, PrintCodes(), uses SETPRC() to send control codes to the printer without affecting PROW() and PCOL() values:

```
#include "Set.ch"
#define ITALICS_ON   CHR(27) + "I"
#define ITALICS_OFF  CHR(27) + "E"
//
SET DEVICE TO PRINTER
@ 12, 10 SAY "This is an"
@ PROW(), PCOL() + 2 SAY PrintCodes(ITALICS_ON) + ;
         "important"
@ PROW(), PCOL() + 2 SAY PrintCodes(ITALICS_OFF) + ;
         "meeting"
SET DEVICE TO SCREEN
RETURN

FUNCTION PrintCodes( cCtrlCode )
   LOCAL nRow, nCol, lPrinter
   lPrinter := SET(_SET_PRINTER, .T.) // SET PRINTER ON
   nRow:= PROW()                      // Save printhead position
   nCol:= PCOL()
   //
   ?? cCtrlCode                       // Send control code
   //
   SETPRC(nRow, nCol)
   SET(_SET_PRINTER, lPrinter)        // Restore printer setting
```

```
        RETURN ""                       // Return a null string
```

Files: Library is CLIPPER.LIB.

See Also: PCOL() PROW() SET DEVICE SET()

SOUNDEX()
Convert a character string to soundex form

Syntax

SOUNDEX(<cString>) --> cSoundexString

Arguments

<cString> is the character string to convert.

Returns

SOUNDEX() returns a four-digit character string in the form A999.

Description

SOUNDEX() is a character function that indexes and searches for sound-alike or phonetic matches. It is used in applications where the precise spelling of character keys is not known or where there is a high probability of misspelled names. Misspelling is common in real-time transaction systems where the data entry operator is receiving information over the telephone. SOUNDEX() works by bringing sound-alikes together under the same key value. Note, however, the soundex method is not absolute. Keys that are quite different can result in the same soundex value.

Examples

. This example builds an index using SOUNDEX() to create the key values. It then searches for a value found in the Salesman field:

```
USE Sales
INDEX ON SOUNDEX(Salesman) TO Salesman
SEEK SOUNDEX("Smith")
? FOUND(), Salesman            // Result: .T. Smith
```

. Here, a search is made for the same key as above but with a different spelling:

```
SEEK SOUNDEX("Smythe")
? FOUND(), Salesman            // Result: .T. Smith
```

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\SOUNDEX.C.

See Also: INDEX LOCATE SEEK SET SOFTSEEK

SPACE()
Return a string of spaces

Syntax

SPACE(<nCount>) --> cSpaces

Arguments

<nCount> is the number of spaces to return, up to a maximum of 65,535 (64K).

Returns

SPACE() returns a character string. If <nCount> is zero, SPACE() returns a null string ("").

Description

SPACE() is a character function that returns a specified number of spaces. It is the same as REPLICATE(" ", <nCount>). SPACE() can initialize a character variable before associating it with a GET. SPACE() can also pad strings with leading or trailing spaces. Note, however, that the PADC(), PADL(), and PADR() functions are more effective for this purpose.

Examples

. This example uses SPACE() to initialize a variable for data input:

```
USE Customer NEW
MEMVAR->Name = SPACE(LEN(Customer->Name))
@ 10,10 SAY "Customer Name" GET MEMVAR->Name
READ
```

Files: Library is CLIPPER.LIB.

See Also: PAD() REPLICATE()

SQRT()
Return the square root of a positive number

Syntax

SQRT(<nNumber>) --> nRoot

Arguments

<nNumber> is a positive number to take the square root of.

Returns

SQRT() returns a numeric value calculated to double precision. The number of decimal places displayed is determined solely by SET DECIMALS regardless of SET FIXED. A negative <nNumber> returns zero.

Description

SQRT() is a numeric function used anywhere in a numeric calculation to compute a square root (e.g., in an expression that calculates standard deviation).

Examples

. These examples show various results of SQRT():

```
SET DECIMALS TO 5
//
? SQRT(2)                 // Result: 1.41421
? SQRT(4)                 // Result: 2.00000
? SQRT(4) ** 2            // Result: 4.00000
? SQRT(2) ** 2            // Result: 2.00000
```

Files: Library is CLIPPER.LIB.

See Also: SET DECIMALS SET FIXED

STR()
Convert a numeric expression to a character string

Syntax

STR(<nNumber>, [<nLength>], [<nDecimals>]) --> cNumber

Arguments

<nNumber> is the numeric expression to convert to a character string.

<nLength> is the length of the character string to return, including decimal digits, decimal point, and sign.

<nDecimals> is the number of decimal places to return.

Returns

STR() returns <nNumber> formatted as a character string. If the optional length and decimal arguments are not specified, STR() returns the character string according to the following rules:

Results of STR() with No Optional Arguments

Expression	Return Value Length
Field Variable	Field length plus decimals
Expressions/constants	Minimum of 10 digits plus decimals
VAL()	Minimum of 3 digits
MONTH()/DAY()	3 digits
YEAR()	5 digits
RECNO()	7 digits

Description

STR() is a numeric conversion function that converts numeric values to character strings. It is commonly used to concatenate numeric values to character strings. STR() has applications displaying numbers, creating codes such as part numbers from numeric values, and creating index keys that combine numeric and character data.

STR() is like TRANSFORM() which formats numeric values as character strings using a mask instead of length and decimal specifications.

The inverse of STR() is VAL() which converts character numbers to numerics.

Notes

. If <nLength> is less than the number of whole number digits in <nNumber>, STR() returns asterisks instead of the number.

. If <nLength> is less than the number of decimal digits required for the decimal portion of the returned string, CA-Clipper rounds the number to the available number of decimal places.

. If <nLength> is specified, but <nDecimals> is omitted (no decimal places), the return value is rounded to an integer.

Examples

. These examples demonstrate the range of values returned by STR(), depending on the arguments specified:

nNumber:= 123.45

```
? STR(nNumber)                // Result:        123.45
? STR(nNumber, 4)             // Result:  123
? STR(nNumber, 2)             // Result: **
? STR(nNumber * 10, 7, 2)     // Result: 1234.50
? STR(nNumber * 10, 12, 4)    // Result:    1234.5000
? STR(nNumber, 10, 1)         // Result:    1234.5
```

. This example uses STR() to create an index with a compound key of order numbers and customer names:

```
USE Customer NEW
INDEX ON STR(NumOrders, 9) + CustName TO CustOrd
```

Files: Library is CLIPPER.LIB.

See Also: TRANSFORM() VAL()

STRTRAN()
Search and replace characters within a character string or memo field

---

Syntax

STRTRAN(<cString>, <cSearch>,
[<cReplace>], [<nStart>], [<nCount>]) --> cNewString

Arguments

<cString> is the character string or memo field to search.

<cSearch> is the sequence of characters to locate.

<cReplace> is the sequence of characters with which to replace <cSearch>. If this argument is not specified, the specified instances of the search argument are replaced with a null string ("").

<nStart> is the first occurrence that will be replaced. If this argument is omitted, the default is one.

<nCount> is the number of occurrences to replace. If this argument is not specified, the default is all.

Returns

STRTRAN() returns a new character string with the specified instances of <cSearch> replaced with <cReplace>.

Description

STRTRAN() is a character function that performs a standard substring search within a character string. When it finds a match, it replaces the search string with the specified replacement string. All instances of <cSearch> are replaced unless <nStart> or <nCount> is specified. Note that STRTRAN() replaces substrings and, therefore, does not account for whole words.

Examples

. This example uses STRTRAN() to establish a postmodern analog to a famous quotation:

```
cString:= "To compute, or not to compute?"
? STRTRAN(cString, "compute", "be")
// Result: To be, or not to be?
```

Files: Library is CLIPPER.LIB.

See Also: AT() HARDCR() MEMOTRAN() RAT() STUFF() SUBSTR()

STUFF()
Delete and insert characters in a string

Syntax

STUFF(<cString>, <nStart>,
<nDelete>, <cInsert>) --> cNewString

Arguments

<cString> is the target character string into which characters are inserted and deleted.

<nStart> is the starting position in the target string where the insertion/deletion occurs.

<nDelete> is the number of characters to delete.

<cInsert> is the string to insert.

Returns

STUFF() returns a copy of <cString> with the specified characters deleted and with <cInsert> inserted.

Description

STUFF() is a character function that deletes <nDelete> characters from <cString> beginning at the <nStart> position. Then, it inserts <cInsert> into the resulting string beginning at <nStart> to form the return string. With this, STUFF() can perform the following six operations:

. **Insert:** If <nDelete> is zero, no characters are removed from <cString>. <cInsert> is then inserted at <nStart>, and the entire string is returned. For example, STUFF("My dog has fleas.", 12, 0, "no ") returns "My dog has no fleas."

. **Replace:** If <cInsert> is the same length as <nDelete>, <cInsert> replaces characters beginning at <nStart>. The same number of characters are deleted as are inserted, and the resulting string is the same length as the original. For example, STUFF("My dog has fleas.", 12, 5, "bones") returns "My dog has bones."

. **Delete:** If <cInsert> is a null string (""), the number of characters specified by <nDelete> are removed from <cString>, and the string is returned without any added characters. For example, STUFF("My dog has fleas.", 1, 3, "") returns "dog has fleas."

. **Replace and insert:** If <cInsert> is longer than <nDelete>, all characters from <nStart> up to <nDelete> are replaced and the rest of <cInsert> is inserted. Since more characters are inserted than are deleted, the resulting string is always longer than the original. For example, STUFF("My dog has fleas.", 8, 3, "does not have") returns "My dog does not have fleas."

. **Replace and delete:** If the length of <cInsert> is less than <nDelete>, more characters are deleted than inserted. The resulting string, therefore, is shorter than the original. For example, STUFF("My dog has fleas.", 8, 3, "is") returns "My dog is fleas."

. **Replace and delete rest:** If <nDelete> is greater than or equal to the number of characters remaining in <cString> beginning with <nStart>, all remaining characters are deleted before <cInsert> is inserted. For example, STUFF("My dog has fleas.", 8, 10, "is.") returns "My dog is."

Examples

. These examples demonstrate the six basic operations of STUFF():

```
// Insert
? STUFF("ABCDEF", 2, 0, "xyz")      // Result: AxyzBCDEF
// Replace
? STUFF("ABCDEF", 2, 3, "xyz")      // Result: AxyzEF
// Delete
? STUFF("ABCDEF", 2, 2, "")         // Result: ADEF
// Replace and insert
? STUFF("ABCDEF", 2, 1, "xyz")      // Result: AxyzCDEF
// Replace and delete
? STUFF("ABCDEF", 2, 4, "xyz")      // Result: AxyzF
// Replace and delete rest
? STUFF("ABCDEF", 2, 10, "xyz")     // Result: Axyz
```

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\STUFF.C.

See Also: AT() LEFT() RAT() RIGHT() STRTRAN() SUBSTR()

SUBSTR()
Extract a substring from a character string

Syntax

SUBSTR(<cString>, <nStart>, [<nCount>]) --> cSubstring

Arguments

<cString> is the character string from which to extract a substring. It can be up to 65,535 (64K) bytes, the maximum character string size in CA-Clipper.

<nStart> is the starting position in <cString>. If <nStart> is positive, it is relative to the leftmost character in <cString>. If <nStart> is negative, it is relative to the rightmost character in the <cString>.

<nCount> is the number of characters to extract. If omitted, the substring begins at <nStart> and continues to the end of the string. If <nCount> is greater than the number of characters from <nStart> to the end of <cString>, the extra is ignored.

Returns

SUBSTR() returns a character string.

Description

SUBSTR() is a character function that extracts a substring from another character string or memo field. SUBSTR() is related to the LEFT() and RIGHT() functions which extract substrings beginning with leftmost and rightmost characters in <cString>, respectively.

The SUBSTR(), RIGHT(), and LEFT() functions are often used with both the AT() and RAT() functions to locate either the first and/or the last position of a substring before extracting it. They are also used to display or print only a portion of a character string.

Examples

. These examples extract the first and last name from a variable:

```
cName:= "Biff Styvesent"
? SUBSTR(cName, 1, 4)               // Result: Biff
? SUBSTR(cName, 6)                  // Result: Styvesent
? SUBSTR(cName, LEN(cName) + 2)     // Result: null string
? SUBSTR(cName, -9)                 // Result: Styvesent
? SUBSTR(cName, -9, 3)              // Result: Sty
```

. This example uses SUBSTR() with AT() and RAT() to create a user-defined function to extract a filename from a file specification:

```
? FileBase("C:\PRG\MYFILE.OBJ")      // Result: MYFILE.OBJ

FUNCTION FileBase( cFile )
   LOCAL nPos
   IF (nPos := RAT("\", cFile)) != 0
      RETURN SUBSTR(cFile, nPos + 1)
   ELSEIF (nPos := AT(":", cFile)) != 0
      RETURN SUBSTR(cFile, nPos + 1)
   ELSE
      RETURN cFile
   ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: AT() LEFT() RAT() RIGHT() STR()

TIME()
Return the system time

Syntax

TIME() --> cTimeString

Returns

TIME() returns the system time as a character string in the form hh:mm:ss. hh is hours in 24-hour format, mm is minutes, and ss is seconds.

Description

TIME() is a time function that displays the system time on the screen or prints it on a report. TIME() is related to SECONDS() which returns the integer value representing the number of seconds since midnight. SECONDS() is generally used in place of TIME() for time calculations.

Examples

. These examples show the results of TIME() used with SUBSTR() to extract the hour, minutes, and seconds digits:

```
? TIME()                      // Result: 10:37:17
? SUBSTR(TIME(), 1, 2)        // Result: 10
? SUBSTR(TIME(), 4, 2)        // Result: 37
? SUBSTR(TIME(), 7, 2)        // Result: 17
```

Files: Library is CLIPPER.LIB.

See Also: DATE() SECONDS() SUBSTR()

TONE()
Sound a speaker tone for a specified frequency and duration

Syntax

TONE(<nFrequency>, <nDuration>) --> NIL

Arguments

<nFrequency> is a positive numeric value indicating the frequency of the tone to sound.

<nDuration> is a positive numeric value indicating the duration of the tone measured in increments of 1/18 of a second. For example, an <nDuration> value of 18 represents one second.

For both arguments, noninteger values are truncated--not rounded--to their integer portion.

Returns

TONE() always returns NIL.

Description

TONE() is a sound function that indicates various program states to the user. These can be error states, boundary conditions, or the end of a time-consuming process. For example, an error state would sound an error tone before alerting the user with a message or interactive dialog box. A boundary condition might indicate that the user is attempting to cursor past the top or bottom of a column in a TBrowse object. A batch process also might indicate its completion with a sound to alert the user, in case the user has turned away from the screen.

TONE() works by sounding the speaker at the specified frequency for the specified duration. The duration is measured in increments of 1/18 of a second. The frequency is measured in hertz (cycles per second). Frequencies of less than 20 are inaudible. The table below shows the frequencies of standard musical notes.

Note: TONE() works only on IBM PC and 100% compatible computers

Table of Musical Notes

Pitch	Frequency	Pitch	Frequency
C	130.80	mid C	261.70
C#	138.60	C#	277.20
D	146.80	D	293.70
D#	155.60	D#	311.10
E	164.80	E	329.60
F	174.60	F	349.20
F#	185.00	F#	370.00
G	196.00	G	392.00
G#	207.70	G#	415.30
A	220.00	A	440.00
A#	233.10	A#	466.20
B	246.90	B	493.90
		C	523.30

Examples

. This example is a beep function that indicates that a batch operation has completed:

```
FUNCTION DoneBeep
```

```
   TONE(300, 1)
   TONE(100, 1)
   TONE(300, 1)
   TONE(100, 1)
   RETURN NIL
```

. This example is a tone sequence that indicates invalid keystrokes or boundary conditions:

```
FUNCTION ErrorBeep
   TONE(100, 3)
   RETURN NIL
```

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM.

See Also: CHR() SET BELL

TRANSFORM()
Convert any value into a formatted character string

Syntax

TRANSFORM(<exp>, <cSayPicture>) --> cFormatString

Arguments

<exp> is the value to format. This expression can be any valid CA-Clipper data type except array, code block, and NIL.

<cSayPicture> is a string of picture and template characters that describes the format of the returned character string.

Returns

TRANSFORM() converts <exp> to a formatted character string as defined by <cSayPicture>.

Description

TRANSFORM() is a conversion function that formats character, date, logical, and numeric values according to a specified picture string that includes a combination of picture function and template strings. TRANSFORM() formats data for output to the screen or the printer in the same manner as the PICTURE clause of the @...SAY command.

. **Function string:** A picture function string specifies formatting rules that apply to the TRANSFORM() return value as a whole, rather than to particular character positions within <exp>. The function string consists of the @ character, followed by one or more additional characters, each of which has a particular meaning (see table below). If a function string is present, the @ character must be the leftmost character of the picture string, and the function string must not contain spaces. A function string may be specified alone or with a template string. If both are present, the function string must precede the template string, and the two must be separated by a single space.

TRANSFORM() Functions

Function	Action
B	Displays numbers left-justified
C	Displays CR after positive numbers
D	Displays date in SET DATE format
E	Displays date in British format
R	Nontemplate characters are inserted
X	Displays DB after negative numbers
Z	Displays zeros as blanks
(	Encloses negative numbers in parentheses
!	Converts alphabetic characters to uppercase

. **Template string:** A picture template string specifies formatting rules on a character by character basis. The template string consists of a series of characters, some of which have special meanings (see table below). Each position in the template string corresponds to a position in the value of the <exp> argument. Because TRANSFORM() uses a template, it can insert formatting characters such commas, dollar signs, and parentheses.

Characters in the template string that have no assigned meanings are copied literally into the return value. If the @R picture function is used, these characters are inserted between characters of the return value; otherwise, they overwrite the corresponding characters

of the return value. A template string may be specified alone or with a function string. If both are present, the function string must precede the template string, and the two must be separated by a single space.

TRANSFORM() Templates

Template	Action
A,N,X,9,#	Displays digits for any data type
L	Displays logicals as "T" or "F"
Y	Displays logicals as "Y" or "N"
!	Converts an alphabetic character to uppercase
$	Displays a dollar sign in place of a leading space in a numeric
*	Displays an asterisk in place of a leading space in a numeric
.	Specifies a decimal point position
,	Specifies a comma position

Examples

. This example formats a number into a currency format using a template:

```
? TRANSFORM(123456, "$999,999")   // Result: $123,456
```

. This example formats a character string using a function:

```
? TRANSFORM("to upper", "@!")     // Result: TO UPPER
```

Files: Library is CLIPPER.LIB.

See Also: @...SAY LOWER() PAD() STR() UPPER() Picture codes

TYPE()
Determine the type of an expression

Syntax

TYPE(<cExp>) --> cType

Arguments

<cExp> is a character expression whose type is to be determined. <cExp> can be a field, with or without the alias, a private or public variable, or an expression of any type.

Returns

TYPE() returns one of the following characters:

TYPE() Return Values

Returns	Meaning
A	Array
B	Block
C	Character
D	Date
L	Logical
M	Memo
N	Numeric
O	Object
U	NIL, local, or static
UE	Error syntactical
UI	Error indeterminate

Description

TYPE() is a system function that returns the type of the specified expression. It can test expression validity as long as the expression uses CLIPPER.LIB functions and does not reference local or static variables, user-defined functions, or built-in functions supplied in EXTEND.LIB.

TYPE() is like VALTYPE() but uses the macro operator (&) to determine the type of the argument. This precludes the use of TYPE() to determine the type of local and static variables. VALTYPE(), by contrast, evaluates an expression and determines the data type of the return value. This lets you determine the type of user-defined functions as well as local and static variables.

Notes

- **Array references:** References to private and public arrays return "A." References to array elements return the type of the element.

- **IF():** To return the appropriate data type for an IF() expression, TYPE() evaluates the condition then returns the type of the evaluated path. If either the IF() condition or the evaluated path are invalid, TYPE() returns "UE."

- **Testing parameters:** TYPE() can only test the validity of parameters received using the PARAMETERS statement. Testing a parameter declared as part of a FUNCTION or PROCEDURE declaration always returns "U." because local parameters do not have a symbol in the symbol table. To determine whether an argument was skipped or left off the end of the argument list, compare the parameter to NIL or use VALTYPE().

- **User-defined and EXTEND.LIB functions:** If a reference is made anywhere in an expression to a function not found in CLIPPER.LIB (a user-defined or EXTEND.LIB function), TYPE() returns "UI." If the user-defined function is not linked into the current program, TYPE() returns "U."

Examples

- These examples demonstrate various results from invocations of TYPE():

```
? TYPE('SUBSTR("Hi There", 4, 5)')      // Result: C
? TYPE("UDF()")                         // Result: UI
? TYPE('IF(.T., "true", 12)')           // Result: C
```

- This example shows two methods for testing for the existence and type of declared parameters:

```
FUNCTION TestParams
   PARAMETERS cParam1, nParam2
   IF cParam1 = NIL
      ? "First parameter was not passed"
      cParam1 := "Default value"
   ENDIF
   //
   IF TYPE(nParam2) == "U"
      ? "Second parameter was not passed"
   ENDIF
   .
   . <statements>
   .
   RETURN NIL
```

Files: Library is CLIPPER.LIB.

See Also: VALTYPE()

UPDATED()
Determine whether a GET changed during a READ

Syntax

UPDATED() --> lChange

Returns

UPDATED() returns true (.T.) if data in a GET is added or changed; otherwise, it returns false (.F.).

Description

UPDATED() determines whether characters were successfully entered into a GET from the keyboard during the most current READ. Each time READ executes, UPDATED() is set to false (.F.). Then, any change to a GET entered from the keyboard sets UPDATED() to true (.T.) after the user successfully exits the GET. If the user presses Esc before exiting the first GET edited, UPDATED() remains false (.F.). Once UPDATED() is set to true (.T.), it retains this value until the next READ is executed.

Within a SET KEY or VALID procedure, you can change the current GET variable using the KEYBOARD command or by assigning a new value with one of the many assignment operators. Changing the variable with KEYBOARD is the same as if the user had entered the change directly from the keyboard, and UPDATED() is set accordingly. However, since UPDATED() reflects only those changes made from the keyboard, an assignment to the GET variable does not affect UPDATED().

Examples

. This example assigns field values from Customer.dbf to variables and edits them. If the user changes any of the values, the field variables for the current record are updated with the new values:

```
USE Customer NEW
CLEAR
MEMVAR->Customer = Customer->Customer
MEMVAR->Address = Customer->Address
@ 1, 1 SAY "Name:" GET MEMVAR->Customer
@ 2, 1 SAY "Address:" GET MEMVAR->Address
READ
//
IF UPDATED()
   Customer->Customer := MEMVAR->Customer
   Customer->Address := MEMVAR->Address
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: @...GET READ SET KEY

UPPER()
Convert lowercase characters to uppercase

Syntax

UPPER(<cString>) --> cUpperString

Arguments

<cString> is the character string to convert.

Returns

UPPER() returns a copy of <cString> with all alphabetical characters converted to uppercase. All other characters remain the same as in the original string.

Description

UPPER() is a character function that converts lower and mixedcase strings to uppercase. It is related to LOWER() which converts upper and mixedcase strings to lowercase. UPPER() is related to the ISUPPER() and ISLOWER() functions which determine whether a string begins with an uppercase or lowercase letter.

UPPER() is generally used to format character strings for display purposes. It can, however, be used to normalize strings for case-independent comparison or INDEXing purposes.

Examples

. These examples illustrate the effects of UPPER():

```
? UPPER("a string")           // Result: A STRING
? UPPER("123 char = <>")      // Result: 123 CHAR = <>
```

. This example uses UPPER() as part of a case-independent condition:

```
USE Customer INDEX CustName NEW
LIST CustName FOR "KATE" $ UPPER(Customer)
```

. UPPER() is also useful for creating case-independent index key expressions:

```
USE Customer NEW
INDEX ON UPPER(Last) TO CustLast
```

. Later, use the same expression to look up Customers:

```
MEMVAR->Last = SPACE(15)
@ 10, 10 GET MEMVAR->Last
READ

SEEK UPPER(MEMVAR->Last)
```

Files: Library is CLIPPER.LIB.

See Also: ISLOWER() ISUPPER() LOWER()

USED()
Determine whether a database file is in USE

Syntax

USED() --> lDbfOpen

Returns

USED() returns true (.T.) if there is a database file in USE; otherwise, it returns false (.F.).

Description

USED() is a database function that determines whether there is a database file in USE in a particular work area. By default, USED() operates on the currently selected work area. It will operate on an unselected work area if you specify it as part of an aliased expression.

Examples

. This example determines whether a database file is in USE in the current work area:

```
USE Customer NEW
? USED()              // Result: .T.
CLOSE
? USED()              // Result: .F.
```

Files: Library is CLIPPER.LIB.

See Also: ALIAS() SELECT SELECT() USE

VAL()
Convert a character number to numeric type

Syntax

VAL(<cNumber>) --> nNumber

Arguments

<cNumber> is the character expression to convert.

Returns

VAL() returns <cNumber> converted to a numeric value including decimal digits.

Description

VAL() is a character conversion function that converts a character string containing numeric digits to a numeric value. When VAL() is executed, it evaluates <cNumber> until a second decimal point, the first non-numeric character, or the end of the expression is encountered. Leading spaces are ignored. When SET FIXED is ON, VAL() returns the number of decimal places specified by SET DECIMALS, rounding <cNumber> if it is specified with more digits than the current DECIMALS value. As with all other functions that round, digits between zero and four are rounded down, and digits between five and nine are rounded up. When SET FIXED is OFF, VAL() returns the number of decimal places specified in <cNumber>.

VAL() is the opposite of STR() and TRANSFORM() which convert numeric values to character strings.

Examples

. These examples illustrate VAL() with SET FIXED ON and SET DECIMALS TO 2:

```
SET DECIMALS TO 2
SET FIXED ON
//
? VAL("12.1234")         // Result:   12.12
? VAL("12.1256")         // Result:   12.13
? VAL("12A12")           // Result:   12
? VAL("A1212")           // Result:      0
? VAL(SPACE(0))          // Result:      0
? VAL(SPACE(1))          // Result:      0
? VAL(" 12.12")          // Result:   12.12
? VAL("12 .12")          // Result:   12.00
```

Files: Library is CLIPPER.LIB.

See Also: ROUND() SET DECIMALS SET FIXED STR() TRANSFORM()

VALTYPE()
Determine the data type returned by an expression

Syntax

VALTYPE(<exp>) --> cType

Arguments

<exp> is an expression of any type.

Returns

VALTYPE() returns a single character representing the data type returned by <exp>. VALTYPE() returns one of the following characters:

VALTYPE() Return Values

Returns	Meaning
A	Array
B	Block
C	Character
D	Date
L	Logical
M	Memo
N	Numeric
O	Object
U	NIL

Description

VALTYPE() is a system function that takes a single argument, evaluates it, and returns a one character string describing the data type of the return value. It is similar to TYPE(), but differs by actually evaluating the specified argument and determining the type of the return value. For this reason, you can determine the type of local and static variables, user-defined functions, and EXTEND.LIB functions. TYPE(), by contrast, uses the macro operator (&) to evaluate the type of its argument. Note that if the argument does not exist an error ("undefined error") will occur, unlike TYPE which will return "U."

Examples

. These examples show the return values for several data types:

```
? VALTYPE(1)                // Result: N
? VALTYPE("GOOB")           // Result: C
? VALTYPE(NIL)              // Result: U
? VALTYPE(array)            // Result: A
? VALTYPE(block)            // Result: B
```

Files: Library is CLIPPER.LIB.

See Also: TYPE()

VERSION()
Returns CA-Clipper version

Syntax

VERSION() --> cVersion

Returns

VERSION() returns the version number of the CA-Clipper library, EXTEND.LIB, as a character value.

Description

VERSION() is an environment function that returns the version of the CA-Clipper library, EXTEND.LIB.

Files: Library is EXTEND.LIB.

WORD()*
Convert CALL command numeric parameters from double to int

Syntax

WORD(<nNumber>) --> NIL

Arguments

<nNumber> is the numeric value to convert to int specified in the range of plus or minus 32,767, inclusive.

Returns

Used as an argument for the CALL command, WORD() returns an integer. In all other contexts, it returns NIL.

Description

WORD() is a numeric conversion function that converts numeric parameters of the CALL command from double to integer values. WORD() is a compatibility command and therefore not recommended. Both the CALL command and the WORD() function are superseded by facilities provided by the Extend System. Refer to the Using the Extend System chapter in the Technical Reference guide for more information.

Examples

. This example uses WORD() as an argument of the CALL command:

```
CALL Cproc WITH WORD(30000), "Some text"
```

Files: Library is CLIPPER.LIB.

YEAR()
Convert a date value to the year as a numeric value

Syntax

YEAR(<dDate>) --> nYear

Arguments

<dDate> is the date value to convert.

Returns

YEAR() returns the year of the specified date value including the century digits as a four-digit numeric value. The value returned is not affected by the current DATE or CENTURY format. Specifying a null date (CTOD("")) returns zero.

Description

YEAR() is a date conversion function that converts a date value to a numeric year value. Use it in calculations for things like periodic reports or for formatting date displays.

YEAR() is a member of a group of functions that return components of a date value as numeric values. The group includes DAY() and MONTH() which return the day and month values as numeric values.

Examples

. These examples illustrate YEAR() using the system date:

```
? DATE()                    // Result: 09/20/90
? YEAR(DATE())              // Result: 1990
? YEAR(DATE()) + 11         // Result: 2001
```

. This example creates a user-defined function using YEAR() to format a date value in the form: month day, year:

```
? Mdy(DATE())               // Result: September 20, 1990

FUNCTION Mdy( dDate )
   RETURN CMONTH(dDate) + " " + ;
         LTRIM(STR(DAY(dDate)));
      + "," + STR(YEAR(dDate))
```

Files: Library is CLIPPER.LIB.

See Also: DAY() MONTH()

?|??
Display one or more values to the console

Syntax

? | ?? [<exp list>]

Arguments

<exp list> is a list of values to display and can be any combination of data types, including memo.

If you specify no argument and use the ? command, a carriage return/linefeed is sent to the console. If you use the ?? command without arguments, nothing happens.

Description

? and ?? are console commands that display the results of one or more expressions, separated by a space, to the console. These commands are also command synonyms for the QOUT() and QQOUT() functions, respectively.

Although functionally similar, ? and ?? differ slightly. ? sends a carriage return/linefeed to the console before displaying the results of the expression list. ?? displays output at the current screen cursor or printhead position. This lets you use successive ?? commands to display output to the same line.

A ? or ?? command locates the cursor or printhead one position to the right of the last character displayed. If SET PRINTER is OFF, ROW() and COL() are updated to reflect the new cursor position. If SET PRINTER is ON, PROW() and PCOL() are updated with the new printhead position.

If output from a ? or ?? command reaches the edge of the screen as reported by MAXCOL(), it wraps to the next line. If the output reaches the bottom of the screen as reported by MAXROW(), the screen scrolls up one line.

You can echo output from the ? or ?? commands to the printer by specifying a SET PRINTER ON command before beginning output. You can echo output from both of these commands to a text file using SET ALTERNATE TO <xcFile> to create the file, and SET ALTERNATE ON to begin echoing to the file. Like other console commands, SET CONSOLE OFF suppresses the display to the screen without affecting output to the printer or text file.

To format any expression specified, use TRANSFORM() or a user-defined function. If you need to pad a variable length value for column alignment, use any of the PAD() functions to left-justify, right-justify, or center the value. See the examples below.

Examples

. This example prints a record from a database file using ? and ?? commands with PADR() to assure column alignment:

```
LOCAL nPage := 0, nLine := 99
USE Salesman INDEX Salesman NEW
SET PRINTER ON
SET CONSOLE OFF
DO WHILE !EOF()
   IF nLine > 55
      IF nPage != 0
         EJECT
      ENDIF
      ? PADR("Page", LTRIM(STR(nPage++)), 72)
```

```
            ?? DTOC(DATE())
            ?
            ?
            ? PADC("Sales Listing", 79)
            ?
            nLine := 5
         ENDIF
         ? Name, Address, PADR(RTRIM(City) + "," ;
               + State, 20), ZipCode
         nLine++
         SKIP
      ENDDO
      SET CONSOLE ON
      SET PRINTER OFF
      CLOSE Salesman
```

Files: Library is CLIPPER.LIB.

See Also: @...SAY PAD() SET CONSOLE SET PRINTER TRANSFORM()

@...BOX
Draw a box on the screen

Syntax

```
@ <nTop>, <nLeft>, <nBottom>, <nRight>
   BOX <cBoxString> [COLOR <cColorString>]
```

Arguments

<nTop>, **<nLeft>**, **<nBottom>**, **<nRight>** define the coordinates of the box. @...BOX draws a box using row values from zero to MAXROW(), and column values from zero to MAXCOL(). If <nBottom> and <nRight> are larger than MAXROW() and MAXCOL(), the bottom-right corner is drawn off the screen.

BOX <cBoxString> defines a string of eight border characters and a fill character. If <cBoxString> is specified as a single character, that character draws the whole box.

COLOR <cColorString> defines the display color of the drawn box. If not specified, the box is drawn using the standard color setting of the current system color as defined by SETCOLOR(). Note that <cColorString> is a character expression containing the standard color setting. If you want to specify a literal color setting, enclose it within quote marks.

Description

@...BOX draws a box on the screen using configurable border and fill characters. @...BOX draws the box using <cBoxString> starting from the upper lefthand corner, proceeding clockwise and filling the screen region with the ninth character. If the ninth character is not specified, the screen region within the box is not painted. Existing text and color remain unchanged.

After @...BOX executes, the cursor is located in the upper corner of the boxed region at <nTop> + 1 and <nLeft> + 1. ROW() and COL() are also updated to reflect the new cursor position.

Examples

. These examples draw two boxes using box manifest constants defined in the supplied header file, Box.ch. The first example draws a box using the specified characters for the border, but leaves all other areas of the screen intact. The second example draws the same box filling the box region with space characters:

```
#include "Box.ch"
// Draw a box with a double-line top with a
// single-line side
@ 1, 1, 22, 79 BOX B_DOUBLE_SINGLE
// Draw the same box filling the box region with
// spaces
@ 1, 1, 22, 79 BOX B_DOUBLE_SINGLE + SPACE(1)
```

Files: Library is CLIPPER.LIB, header file is Box.ch.

See Also: @...CLEAR @...TO DISPBOX() SCROLL() Box characters Colors

@...CLEAR
Clear a rectangular region of the screen

Syntax

```
@ <nTop>, <nLeft> [CLEAR
   [TO <nBottom>, <nRight>]]
```

Arguments

<nTop> and **<nLeft>** define the upper left corner coordinate.

TO <nBottom>, <nRight> defines the lower right corner coordinates of the screen region to CLEAR. If the TO clause is not specified, these coordinates default to MAXROW() and MAXCOL().

Description

@...CLEAR erases a rectangular region of the screen by filling the specified region with space characters using the current standard color setting. After @...CLEAR erases the designated region, the cursor is located in the upper corner of the region at <nTop> + 1 and <nLeft> + 1. ROW() and COL() are also updated to reflect the new cursor position.

Examples

. This example erases the screen from 10, 10 to 20, 40, painting the region blue then displaying a box bright cyan on blue:

```
SETCOLOR("BG+/B")
@ 10, 10 CLEAR TO 20, 40
@ 10, 10 TO 20, 40
```

Files: Library is CLIPPER.LIB.

See Also: @...BOX CLEAR SCREEN SCROLL() SETCOLOR()

@...GET
Create a new Get object and display it to the screen

Syntax

```
@ <nRow>, <nCol>
   [SAY <exp>
      [PICTURE <cSayPicture>]
      [COLOR <cColorString>]]
   GET <idVar>
      [PICTURE <cGetPicture>]
      [COLOR <cColorString>]
      [WHEN <lPreExpression>]
      [RANGE* <dnLower>, <dnUpper>] |
      [VALID <lPostExpression>]
```

Arguments

<nRow> and **<nCol>** specify the row and column coordinates for the operation. If the SAY clause is present, <nRow> and <nCol> specify the coordinates for the SAY, and the GET is displayed to the right of the SAY output. If the SAY clause is not present, <nRow> and <nCol> directly specify the coordinates for the GET. In either case, output which extends beyond the visible extent of the display is clipped and does not appear.

SAY <exp> displays the value of <exp> at the specified coordinates. If you specify the PICTURE <cSayPicture> clause, <exp> is formatted according to the rules of SAY pictures.

GET <idVar> specifies the name of the variable associated with the GET. <idVar> may be of any storage class (if the storage class is ambiguous, FIELD is assumed). If <idVar> contains an array value, you must supply one or more subscripts. The current value of <idVar> is displayed at the GET coordinates. The value must be character, date, numeric, or logical type. Array, NIL, code block, and null string values are not permitted.

PICTURE <cGetPicture> specifies the display format and editing rules for the GET.

COLOR <cColorString> defines the color settings for the current Get object. <cColorString> is a character expression containing one or more color settings. You must specify literal color settings enclosed in quote marks.

If you specify two color settings, the first determines the unselected color (the display color for the Get object) and the second determines the selected color (the color when the Get object has focus). If you only specify one color, it determines both the unselected and selected colors for the Get object.

On a combined @...SAY...GET command, two COLOR clauses are required to specify colors for both the SAY and the GET: one for the SAY, and one for the GET.

WHEN <lPreExpression> specifies an expression that must be satisfied before the cursor can enter the GET during a READ. The expression can optionally be a code block. The GET object is passed as a parameter to the code block.

VALID <lPostExpression> specifies an expression that must be satisfied before the cursor can leave the GET during a READ. The expression can optionally be a code block. The GET object is passed as a parameter to the code block.

RANGE* <dnLower>, <dnUpper> specifies a range of allowable

values for input to the GET. During a READ command, if you enter a new GET value that does not fall within the specified range, the cursor cannot leave the GET.

Description

The @...GET command creates a new Get object, displays its value, and adds it to the array referred to by the variable GetList. If no variable called GetList has been declared or created in the current procedure, and no PRIVATE variable called GetList exists from a previous procedure, the system uses the predefined PUBLIC variable GetList. A subsequent READ command activates the GETs contained in the GetList array and allows the user to edit their contents.

Each Get object has an associated variable, <idVar>. The variable may be of any storage class, including a database field, private, public, local, or static variable. If <idVar> is followed by one or more subscripts, the specified array element is associated with the GET. When the Get object is created, the <idVar> name is stored in the Get object, along with a code block which allows the value of <idVar> to be retrieved or assigned during the READ.

The READ command performs a full-screen edit of the GETs in the GetList array. As the user moves the cursor into each GET, the value of the associated <idVar> is retrieved by evaluating the code block saved in the Get object. The value is converted to textual form and placed in a buffer within the Get object. This buffer is displayed on the screen, and the user is allowed to edit the text from the keyboard. When the user moves the cursor out of the GET, the updated buffer is converted back to the appropriate data type and assigned to <idVar>.

Automatic formatting and validation: During a READ, some formatting and edit validation is automatically performed for numeric, date, and logical values. As the user is typing, an automatic data type test is performed on each key pressed, preventing the user from entering an invalid character.

Prevalidation: The WHEN clause specifies a condition which must be satisfied before the cursor can enter the GET. During a READ, <lPreExpression> is evaluated whenever the user attempts to move the cursor into the GET. If it evaluates to true (.T.), the cursor can enter; otherwise, the GET is skipped.

Postvalidation: You may perform postvalidation using either the VALID or RANGE* clauses. VALID specifies a condition which must be satisfied before the cursor can leave the GET. During a READ, <lPostExpression> is evaluated whenever the user attempts to move the cursor out of the GET. If it evaluates to true (.T.), the cursor can leave; otherwise, the cursor remains in the GET. RANGE* specifies a range of acceptable values for numeric or date values. If the value entered by the user is not within the specified range, the cursor cannot leave the GET.

PICTURE: When you specify the PICTURE clause for a GET, the character string specified by <cGetPicture> controls formatting and edit validation. The picture string controls the display format like a SAY picture. It also controls the way the user can edit the buffer. A picture string consists of two distinct parts, a function string and a template string, either or both of which may be present.

. **Function string:** A PICTURE function string specifies formatting or validation rules which apply to the GET's display value as a whole, rather than to particular character positions within it. The function string consists of the @ character, followed by one or more additional characters, each of which has a particular meaning (see table below). The function string must be the first element of a PICTURE clause and cannot contain spaces. A function string may be specified alone or with a template string. If both are present, the

function string must precede the template string, and the two must be separated by a single space.

GET PICTURE Format Functions

Func	Type	Action
A	C	Allows only alphabetic characters
B	N	Displays numbers left-justified
C	N	Displays CR after positive numbers
D	D,N	Displays dates in SET DATE format
E	D,N	Displays dates with day and month inverted independent of the current DATE SETting, numerics with comma and period reverse (European style)
K	ALL	Deletes default text if first key is not a cursor key
R	C	Nontemplate characters are inserted in the display but not saved in the variable
S<n>	C	Allows horizontal scrolling within a GET. <n> is an integer that specifies the width of the region.
X	N	Displays DB after negative numbers
Z	N	Displays zero as blanks
(	N	Displays negative numbers in parentheses with leading spaces
)	N	Displays negative numbers in parentheses without leading spaces
!	C	Converts alphabetic character to uppercase

. **Template string:** A PICTURE template string specifies formatting or validation rules on a character by character basis. The template string consists of a series of characters, some of which have special meanings (see table below). Each position in the template string corresponds to a position in the displayed GET value. Characters in the template string which do not have assigned meanings are copied verbatim into the displayed GET value. If you use the @R picture function, these characters are inserted between characters of the display value, and are automatically removed when the display value is reassigned to <idVar>; otherwise, they overwrite the corresponding characters of the display value and also affect the value assigned to <idVar>. You may specify a template string alone or with a function string. If you use both, the function string must precede the template string, and the two must be separated by a single space.

GET PICTURE Template Symbols

Template	Action
A	Allows only alphabetic characters
N	Allows only alphabetic and numeric characters
X	Allows any character
9	Allows digits for any data type including sign for numerics
#	Allows digits, signs and spaces for any data type
L	Allows only T, F, Y or N
Y	Allows only Y or N
!	Converts an alphabetic character to uppercase
$	Displays a dollar sign in place of a leading space in a numeric
*	Displays an asterisk in place of a leading space in a numeric
.	Displays a decimal point
,	Displays a comma

SCOREBOARD: If a new value is rejected because it fails the RANGE* test or because it is a malformed date value, a message appears on the screen. The message displays in the SCOREBOARD area, which you can

enable or disable using the SET SCOREBOARD command.

Exit with Esc: If the user exits a GET by pressing Esc, the GET variable is restored to the value it had on entry to the GET, and the READ is terminated. No postvalidation is performed. You can enable or suppress this behavior with the SET ESCAPE command.

SET KEY procedures: The SET KEY command lets you specify a procedure to be executed whenever a specific key is pressed during a READ. After a SET KEY procedure terminates, the GET is reactivated with the cursor restored to its previous position.

Lifetime of a Get object: Get objects, like arrays, exist as long as there are active references to them somewhere in the current program. Normally, only the array in the GetList variable maintains a reference to the Get object; the GET is released when GetList is released or reassigned. The CLEAR and CLEAR GETS commands assign an empty array to GetList, as does the READ command unless you specify the SAVE clause.

Assignment: Each GET is associated with a variable, <idVar>, in the @...GET command. At various times during the editing process, <idVar> may be assigned the current value of the Get object's buffer. This occurs in the following instances:

. After the user presses an exit key and before the validation expression is executed

. After the user presses a SET KEY

Also, the current Get object's buffer is refreshed from <idVar> and redisplayed at various intervals:

. After a SET KEY procedure terminates

. After a WHEN expression is evaluated

. After a VALID expression is evaluated

This lets you explicitly assign <idVar> within any of these operations. See the note below for more information.

GET coordinates and display: When you create a Get object using the @...GET command, the row and column coordinates at which the GET is initially displayed are stored into the Get object. When the @...GET command executes, the new GET displays at <nRow> and <nCol>, unless you specify the SAY clause which positions the GET so there is one display column between the last character of the SAY output and the first character of the GET (or of the DELIMITERS, see below).

If SET DELIMITERS is ON when the @...GET command executes , the current DELIMITER characters display on either side of the initial GET display, and the column coordinate of the GET is adjusted accordingly. Note that the delimiters are not attributes of the GET object, but simply display as the SAY clause does.

If INTENSITY is ON, GETs initially display in the current unselected color (or the enhanced color, if no unselected color has been specified). During a READ, the active GET displays in the enhanced color, while the remaining GETs display in the unselected color. With INTENSITY OFF, all GETs display in the standard color.

When a GET displays, the width of the displayed value is determined by the length of the value in <idVar> or, if you specify the PICTURE clause, by the number of positions in <cGetPicture>. If you specify the @S function as a part of <cGetPicture>, the @S argument controls the width of the displayed value.

See Also: @...GET...

ACLONE()
Duplicate a nested or multidimensional array

Syntax

ACLONE(<aSource>) --> aDuplicate

Arguments

<aSource> is the array to duplicate.

Returns

ACLONE() returns a duplicate of <aSource>.

Description

ACLONE() is an array function that creates a complete duplicate of the <aSource> array. If <aSource> contains subarrays, ACLONE() creates matching subarrays and fills them with copies of the values in the <aSource> subarrays. ACLONE() is similar to ACOPY(), but ACOPY() does not duplicate nested arrays.

Examples

. This example creates an array then duplicates it using ACLONE(). The first array is then altered, but the duplicate copy is unaffected:

```
LOCAL aOne, aTwo
aOne := { 1, 2, 3 }        // Result: aOne is {1, 2, 3}
aTwo := ACLONE(aOne)       // Result: aTwo is {1, 2, 3}
aOne[1] := 99              // Result: aOne is {99, 2, 3}
                           // aTwo is still {1, 2, 3}
```

Files: Library is CLIPPER.LIB.

See Also: ACOPY() ADEL() AINS() ASIZE()

Notes

- **WHEN and VALID:** The expressions specified in the WHEN and VALID clauses may be of arbitrary complexity and may include calls to user-defined functions. This is useful for attaching automatic actions to the activation or deactivation of a GET.

- **Assigning <idVar>:** Because of the automatic refresh and display properties of a Get object while it is being READ, you can make an explicit assignment to the Get object's <idVar> within a WHEN or VALID expression. You can directly assign the variable by name in the validation expression or, for private, public, local, or static variables, by passing a reference to <idVar> to a function; the function can then assign <idVar> by assigning the corresponding formal parameter. If <idVar> is a field, it is globally visible and can be assigned by name in a function called by the validation expression.

  When including a Get in a called function, do not include an <idVar> with the same name as a field <idVar>. Field references have precedence over public variables so the public <idVar> will be ignored.

- **GET specific help:** You can use a SET KEY procedure to display help text associated with a Get object. Within the SET KEY procedure, use the READVAR() function to determine the <idVar> associated with the current Get object. Use this information to display the appropriate help text. Remember that when a CA-Clipper-compiled program loads, the F1 KEY is automatically SET TO a procedure or user-defined function named Help.

- **SET DEVICE TO PRINTER:** SET DEVICE TO PRINTER does not direct display of a Get object under the @...GET command to the printer or file.

Examples

- This example uses the VALID clause to validate input into a GET:

```
LOCAL nNumber := 0
@ 10, 10 SAY "Enter a number:" ;
   GET nNumber VALID nNumber > 0
```

- This example demostrates passing a code block with the VALID clause. The parameter oGet is the current get object. Udf() changes the value of the GET:

```
LOCAL GetList := {}, cVar := SPACE(10)
CLS
@ 10, 10 GET cVar  VALID { |oGet| Udf1( oGet ) }
READ
.
.
.
* Udf( <oGet> ) --> .T.

FUNCTION Udf1( oGet )

IF "test" $ oGet:BUFFER          // Compare buffer contents
oGet:varPut( "new value " )      // Change contents
ENDIF

RETURN .T.
```

- This example uses the WHEN clause to prohibit entry into GETs based on the value of another GET. In this example, entering Y in

the Insured field indicates the client has insurance and the user is allowed to enter insurance information. If the client does not have insurance, the cursor moves to the Accident field:

```
@ 10, 10 GET Insured PICTURE "Y"
@ 11, 10 GET InsNumber WHEN Insured
@ 12, 10 GET InsCompany WHEN Insured
@ 13, 10 GET Accident PICTURE "Y"
READ
```

. This is an example of a GET in a secondary work area:

```
USE Invoice NEW
APPEND BLANK
USE Inventory NEW
@ 1, 1 GET Invoice->CustNo
READ
```

. This example uses the @K function to suggest a default input value, but deletes it if the first key pressed is not a cursor key or Return:

```
LOCAL cFile := "Accounts"
@ 1, 1 SAY "Enter file" GET cFile PICTURE "@K"
READ
```

. This is an example of a nested READ using GetList and lexical scoping:

```
#include "inkey.ch"
//
// Local to this function only
LOCAL GetList := {}
LOCAL cName   := SPACE( 10 )
//
CLS
SETKEY( K_F2, { || MiscInfo() } )   // Hot key to special READ
//
// Get object added to Getlist
// works on local getlist
@ 10, 10 SAY "Name" GET cName
READ
//
RETURN NIL

/***
*  MiscInfo() ---> NIL
*/
FUNCTION MiscInfo()
//
LOCAL GetList    := {}               // Local to this
LOCAL cExtraInfo := SPACE( 30 )      // function only
//
// Get object added to getlist
// works on local getlist
@ 12, 10 SAY "Note: " GET cExtraInfo
READ
//
RETURN NIL
```

Files: Library is CLIPPER.LIB.

See Also: @...SAY COL() READ SETCOLOR() Picture codes Colors

@...PROMPT
Paint a menu item and define a message

Syntax

```
@ <nRow>, <nCol> PROMPT <cMenuItem>
   [MESSAGE <cExpression>]
```

Arguments

<nRow> and **<nCol>** are the row and column coordinates of the menu item display. Row values can range from zero to MAXROW(), and column values can range from zero to MAXCOL().

PROMPT <cMenuItem> is the menu item string to display.

MESSAGE <cExpression> defines the message to display each time the current menu item is highlighted. A code block evaluating to a character expression may optionally be used.

Description

@...PROMPT is the display portion of the CA-Clipper lightbar menu system. Each @...PROMPT command paints a menu item in the current standard color and defines an associated MESSAGE to be displayed on the line specified by SET MESSAGE. The lightbar menu is then invoked with MENU TO. You can specify menu items in any order and configuration of row and column position. MENU TO, however, navigates the current list of menu items in the order they were defined. You can define up to 4096 menu items for each menu.

After each @...PROMPT command, the cursor is located one column position to the right of the last menu item character and ROW() and COL() are updated to reflect the new cursor position. This lets you use ROW() and COL() to specify consecutive menu item positions relative to the first one painted. See the example below.

Examples

. This example displays a lightbar menu with the associated messages displayed on the next line. When the user presses Return, the position of the item in the list of menu items is assigned to nChoice:

```
LOCAL nChoice := 1
SET WRAP ON
SET MESSAGE TO 2
@ 1, 3 PROMPT "File" MESSAGE "Access data files"
@ ROW(), COL() + 2 PROMPT "Edit" ;
   MESSAGE "Edit current record"
MENU TO nChoice
```

. This example shows how to pass the MESSAGE clause a code block which calls a user-defined function that displays a message in a different color:

```
SET COLOR TO gr+/b,r+/n
SET MESSAGE TO 23       // This is necessary.
CLEAR SCREEN

@ 3,4 PROMPT "one "  MESSAGE {||UDF("Message One  ")}
@ 4,4 PROMPT "two "  MESSAGE {||UDF("Message Two  ")}
@ 5,4 PROMPT "three" MESSAGE {||UDF("Message Three")}

MENU TO test

FUNCTION UDF(cParm)
```

```
cOldColor := SETCOLOR("w+/n")
@ 22,1 SAY cParm         // Must be different row than the
                         // SET MESSAGE TO nRow
SETCOLOR(cOldColor)
RETURN ""                // Character string must be returned
```

Files: Library is CLIPPER.LIB.

See Also: ACHOICE() MENU TO SET MESSAGE SET WRAP SETCOLOR()

ACOPY()
Copy elements from one array to another

Syntax

ACOPY(<aSource>, <aTarget>,
[<nStart>], [<nCount>], [<nTargetPos>]) --> aTarget

Arguments

<aSource> is the array to copy elements from.

<aTarget> is the array to copy elements to.

<nStart> is the starting element position in the <aSource> array.
If not specified, the default value is one.

<nCount> is the number of elements to copy from the <aSource> array beginning at the <nStart> position. If <nCount> is not specified, all elements in <aSource> beginning with the starting element are copied.

<nTargetPos> is the starting element position in the <aTarget> array to receive elements from <aSource>. If not specified, the default value is one.

Returns

ACOPY() returns a reference to the target array, <aTarget>.

Description

ACOPY() is an array function that copies elements from the <aSource> array to the <aTarget> array. The <aTarget> array must already exist and be large enough to hold the copied elements. If the <aSource> array has more elements, some elements will not be copied.

ACOPY() copies values of all data types including NIL and code blocks. If an element of the <aSource> array is a subarray, the corresponding element in the <aTarget> array will contain a reference to the subarray. Thus, ACOPY() will not create a complete duplicate of a multidimensional array. To do this, use the ACLONE() function.

Examples

. This example creates two arrays, each filled with a value. The first two elements from the source array are then copied into the target array:

```
LOCAL nCount := 2, nStart := 1, aOne, aTwo
aOne := { 1, 1, 1 }
aTwo := { 2, 2, 2 }
ACOPY(aOne, aTwo, nStart, nCount)
// Result: aTwo is now { 1, 1, 2 }
```

Files: Library is CLIPPER.LIB.

See Also: ACLONE() ADEL() AEVAL() AFILL() AINS() ASORT()

@...SAY
Display data at a specified screen or printer row and column

Syntax

```
@ <nRow>, <nCol>
   SAY <exp> [PICTURE <cSayPicture>]
   [COLOR <cColorString>]
```

Arguments

<nRow> and **<nCol>** are the row and column coordinates of the display. Row values can range from zero to a maximum of MAXROW() if the current DEVICE is the SCREEN or 32,766 if the current DEVICE is the PRINTER. Also, column values can range from zero to a maximum of MAXCOL() or 32,766 if the current DEVICE is the PRINTER.

SAY <exp> displays the result of a character, date, logical, or numeric expression to the current DEVICE.

PICTURE <cSayPicture> defines the formatting control for the display of <exp>. CA-Clipper provides two mechanisms, functions and templates, to control formatting. Functions apply to an entire SAY, while templates format characters position by position.

COLOR <cColorString> defines the display color of <exp>. If not specified, <exp> displays in the standard color as defined by SETCOLOR(). <cColorString> is a character expression containing the standard color setting. If you specify a literal color setting, it must be enclosed in quote marks.

On a combined @...SAY...GET command, two COLOR clauses are required to specify colors for both the SAY and the GET: one for the SAY and one for the GET.

Description

@...SAY is a full-screen command that outputs the results of <exp> to either the screen or the printer at the specified row and column coordinates. It can optionally format output using the PICTURE clause. @...SAY creates data entry screens or reports that can be sent to the screen or printer.

When an @...SAY command executes , the output from <exp> is sent to the current device defined with SET DEVICE. The current DEVICE can be the SCREEN or PRINTER. Unlike console commands, @...SAY output to the printer is not echoed to the screen and SET CONSOLE has no effect on @...SAY output to the screen.

If the current DEVICE is the SCREEN (the system default), @...SAY displays output to the screen leaving the cursor one column position to the right of the last character displayed. ROW() and COL() are then updated with this position. Output that displays off the screen, as defined by MAXROW() and MAXCOL(), is clipped and the cursor is positioned beyond the visible screen. All @...SAY output displays are in standard color. Refer to the SETCOLOR() reference in this chapter for more information on color.

If the current DEVICE is set to PRINTER, the display is directed to the printer at the specified <nRow> and <nCol> position. If current MARGIN value is greater than zero, it is added to <nCol> first. The printhead is then advanced one column position to the right of the last character output and PROW() and PCOL() are updated. @...SAY commands to the printer behave differently than to the screen if output is addressed to a printer row or column position less than the current PROW() and PCOL() values.

- If <nRow> is less than PROW(), an automatic EJECT (CHR(12)) is sent to the printer followed by the number of linefeed characters (CHR(10)) required to position the printhead on <nRow> on the following page.

- If <nCol> including the SET MARGIN value is less than PCOL(), a carriage return character (CHR(13)) and the number of spaces required to position <exp> at <nCol> are sent to the printer.

To override this behavior and send control codes to the printer, or for any other reason, you can use SETPRC() to reset PROW() and PCOL() to new values. See the SETPRC() function reference for more information.

If the current DEVICE is the PRINTER, redirect output from @...SAY commands to a file using the SET PRINTER TO <xcFile> command.

@...SAY command output can be formatted using the PICTURE clause with a <cSayPicture>. This performs the same action as the TRANSFORM() function. A <cSayPicture> may consist of a function and/or a template. A PICTURE function imposes a rule on the entire @...SAY output. A PICTURE template defines the length of the @...SAY output and the formatting rule for each position within the output.

- **Function string**: A PICTURE function string specifies formatting rules which apply to the SAY's entire display value, rather than to particular character positions within it. The function string consists of the @ character, followed by one or more additional characters, each of which has a particular meaning (see table below). The function string must not contain spaces. A function string may be specified alone or with a template string. If both are present, the function string must precede the template string, and the two must be separated by a single space.

  SAY and TRANSFORM() PICTURE Format Functions

Function	Action
B	Displays numbers left-justified
C	Displays CR after positive numbers
D	Displays dates in SET DATE format
E	Displays dates and numbers in British format
R	Nontemplate characters are inserted
X	Displays DB after negative numbers
Z	Displays zeros as blanks
(	Encloses negative numbers in parentheses
!	Converts alphabetic characters to uppercase

- **Template string:** A PICTURE template string specifies formatting rules on a character-by-character basis. The template string consists of a series of characters, some of which have special meanings (see table below). Each position in the template string corresponds to a position in the displayed SAY value. Characters in the template string which do not have assigned meanings are copied verbatim into the displayed SAY value. If you use the @R picture function, characters without special PICTURE template string meaning are inserted between characters of the display value; otherwise, they overwrite the corresponding characters of the display value. You may specify a template string alone or with a function string. If both are present, the function string must precede the template string, and the two must be separated by a single space.

  SAY and TRANSFORM() Template Symbols

Template	Action
A,N,X,9,#	Displays digits for any data type
L	Displays logicals as "T" or "F"

```
Y            Displays logicals as "Y" or "N"
!            Converts alphabetic characters to uppercase
$            Display a dollar sign in place of a leading space in a
             number
*            Display an asterisk in place of a leading space in a
             number
.            Specifies a decimal point position
,            Specifies a comma position
---------------------------------------------------------------------
```

Examples

. This example uses an @...SAY with a PICTURE clause to display formatted output:

```
nNetIncome = 7125.50
nNetLoss = -125.50
cPhone = "2134567890"
cName = "Kate Mystic"
//
@ 1, 1 SAY nNetIncome PICTURE "@E 9,999.99"
// Result: .125,50

@ 2, 1 SAY nNetLoss PICTURE "@)"
// Result: (125.50)

@ 3, 1 SAY cPhone PICTURE "@R (999)999-9999"
// Result: (213)456-7890

@ 4, 1 SAY cName PICTURE "@!"
// Result: KATE MYSTIC
```

. This example is a small label printing program that uses SET DEVICE to direct output to the printer and SETPRC() to suppress automatic EJECTs:

```
USE Salesman INDEX Salesman NEW
SET DEVICE TO PRINTER
DO WHILE !EOF()                  // Print all records
   @ 2, 5 SAY RTRIM(FirstName) + ", " + LastName
   @ 3, 5 SAY Street
   @ 4, 5 SAY RTRIM(City) + ", " + State + "  " + ;
             PostalCode
   @ 6, 0 SAY SPACE(1)           // Move to label bottom
   SETPRC(0, 0)                  // Suppress page eject
   SKIP                          // Next record
ENDDO
SET DEVICE TO SCREEN
CLOSE Salesman
```

Files: Library is CLIPPER.LIB.

See Also: @...GET COL() SET DEVICE Picture codes Colors

@...TO
Draw a single or double-line box

Syntax

```
@ <nTop>, <nLeft>
   TO <nBottom>, <nRight> [DOUBLE] [COLOR
      <cColorString>]
```

Arguments

<nTop>, **<nLeft>**, **<nBottom>**, and **<nRight>** define the coordinates of the box. @...TO draws the box using row values from zero to MAXROW() and column values from zero to MAXCOL(). <nBottom> and <nRight> can be larger than the screen size but output is clipped at MAXROW() and MAXCOL().

DOUBLE draws the box with a double line. If not specified, the box is drawn with a single line.

COLOR <cColorString> defines the display color of the drawn box. If not specified, the box is drawn using the standard color setting of the current system color as defined by SETCOLOR(). Note that <cColorString> is a character expression containing the standard color setting. If you specify a literal color setting, enclose it within quote marks.

Description

@...TO draws a single or double-line box on the screen. If <nTop> and <nBottom> are the same, a horizontal line is drawn. If <nLeft> and <nRight> are the same, a vertical line is drawn.

After @...TO finishes drawing, the cursor is located in the upper left corner of the boxed region at <nTop> + 1 and <nLeft> + 1. ROW() and COL() are also updated to reflect the new cursor position.

@...TO is like @...BOX except that @...BOX lets you define the characters of the box and supports a fill character. @...TO, however, is recommended for portability since it does not require the specification of hardware-dependent graphics characters.

Examples

. This example erases a region of the screen, then draws a box of the same size:

```
@ 10, 10 CLEAR TO 20, 40
@ 10, 10 TO 20, 40 DOUBLE COLOR "BG+/B"
```

Files: Library is CLIPPER.LIB.

See Also: @...BOX @...CLEAR DISPBOX() Colors

ACCEPT
Place keyboard input into a memory variable

Syntax

ACCEPT [<expPrompt>] TO <idVar>

Arguments

<expPrompt> is an optional prompt displayed before the input area. The prompt can be an expression of any data type.

<idVar> is the variable that will hold input from the keyboard. If the specified <idVar> does not exist or is not visible, a private variable is created.

Description

ACCEPT is a console command and wait state that takes input from the keyboard and assigns it as a character string to the specified variable. When ACCEPT is executed, it first performs a carriage return/linefeed, displays the prompt, and then begins taking characters from the keyboard at the first character position following the prompt. You may input up to 255 characters. When input reaches the edge of the screen, as defined by MAXCOL(), the cursor moves to the next line.

ACCEPT supports only two editing keys: Backspace and Return. Esc is not supported. Backspace deletes the last character typed. Return confirms entry and is the only key that can terminate an ACCEPT. If Return is the only key pressed, ACCEPT assigns a null value ("") to <idVar>.

Examples

. This example uses ACCEPT to get keyboard input from the user:

```
LOCAL cVar
ACCEPT "Enter a value: " TO cVar
IF cVar == ""
   ? "User pressed Return"
ELSE
   ? "User input:", cVar
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: @...GET @...SAY INKEY() INPUT KEYBOARD WAIT*

APPEND BLANK
Add a new record to the current database file

Syntax

APPEND BLANK

Description

APPEND BLANK is a database command that adds a new record to the end of the current database file and then makes it the current record. The new field values are initialized to the empty values for each data type: character fields are assigned with spaces; numeric fields are assigned zero; logical fields are assigned false (.F.); date fields are assigned CTOD(""); and memo fields are left empty.

If operating under a network with the current database file shared, APPEND BLANK attempts to add and then lock a new record. If another user has locked the database file with FLOCK() or locked LASTREC() + 1 with RLOCK(), NETERR() returns true (.T.). Note that a newly APPENDed record remains locked until you lock another record or perform an UNLOCK. APPEND BLANK does not release an FLOCK() set by the current user.

Examples

. This example attempts to add a record to a shared database file and uses NETERR() to test whether the operation succeeded:

```
USE Sales SHARED NEW
.
. <statements>
.
APPEND BLANK
IF !NETERR()
   <update empty record>...
ELSE
   ? "Append operation failed"
   BREAK
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: APPEND FROM FLOCK() NETERR() RLOCK()

APPEND FROM
Import records from a (.dbf) or ASCII file

Syntax

```
APPEND FROM <xcFile>
   [FIELDS <idField list>]
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
   [SDF | DELIMITED [WITH BLANK | <xcDelimiter>]]
```

Arguments

FROM <xcFile> specifies the name of the source file. You can specify <xcFile> either as a literal filename or as a character expression enclosed in parentheses. If a file extension is not specified, (.dbf) is the default input file type. If SDF or DELIMITED is specified, the file extension is assumed to be (.txt) unless otherwise specified.

FIELDS <idField list> specifies the list of fields to copy from <xcFile>. The default is all fields.

<scope> is the portion of the source database file to APPEND FROM. NEXT <n> APPENDs the first <n> records. RECORD <n> APPENDs only record number <n> from <xcFile>. The default scope is ALL records in <xcFile>.

WHILE <lCondition> specifies the set of records meeting the condition from the first record in the source file until the condition fails.

FOR <lCondition> specifies the conditional set of records to APPEND FROM within the given scope.

SDF identifies a System Data Format ASCII file. Records and fields are fixed length.

DELIMITED identifies an ASCII text file, where character fields are enclosed in double quote marks (the default delimiter). Note that delimiters are not required and CA-Clipper correctly APPENDs character fields not enclosed in them. Fields and records are variable length.

DELIMITED WITH BLANK identifies an ASCII text file in which fields are separated by one space and character fields are not enclosed in delimiters.

DELIMITED WITH <xcDelimiter> identifies a delimited ASCII text file, where character fields are enclosed using the specified delimiter. you can specify <xcDelimiter> as a literal character or as a character expression enclosed in parentheses.

See the tables below for more information regarding the format specification requirements for ASCII text files that you want to APPEND using these arguments.

Warning! If the DELIMITED WITH clause is specified on a COPY TO or APPEND FROM command line, it must be the last clause specified.

Description

APPEND FROM adds records to the current database file from an ASCII text file or another database file. Only fields with the same names and types are APPENDed. Fields with the same name from both the current database file and <xcFile> must be the same data type. If they are not, a runtime error occurs when the APPEND FROM command is invoked.

Any date information in <xcFile> must be in the format yyyymmdd to be properly APPENDed.

In a network environment, APPEND FROM does not require that the current database file be USEed EXCLUSIVEly or locked with FLOCK() to perform its operation. As each record is added, CA-Clipper automatically arbitrates contention for the new record.

When you invoke APPEND FROM, CA-Clipper attempts to open <xcFile> shared and read-only. If access is denied, APPEND FROM terminates with a runtime error. Refer to the Network Programming chapter in the Programming and Utilities guide for more information. No error is raised if you attempt to open a (.dbf) file that is already open.

This table shows the format specifications for SDF text files.

SDF Text File Format Specifications

File Element	Format
Character fields	Padded with trailing blanks
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Padded with leading blanks or zeros
Field separator	None
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

This table shows the format specifications for DELIMITED and DELIMITED WITH <xcDelimiter> ASCII text files.

DELIMITED Text File Format Specifications

File Element	Format
Character fields	May be delimited, with trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros may be truncated
Field separator	Comma
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

This table shows the format specifications for DELIMITED WITH BLANK ASCII text files.

DELIMITED WITH BLANK Text File Format Specifications

File Element	Format
Character fields	Not delimited, trailing blanks may be truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros may be truncated
Field separator	Single blank space
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

Notes

- **Deleted records:** If DELETED is OFF, deleted records in <xcFile> are APPENDed to the current database file and retain their deleted status. If DELETED is ON, however, none of the deleted <xcFile> records is APPENDed.

- **Unmatched field widths:** If a field in the current database file is character type and has a field length greater than the incoming <xcFile> data, CA-Clipper pads the <xcFile> data with blanks. If the current field is character data type and its field length is less than the incoming <xcFile> data, the <xcFile> data is truncated to fit. If the current field is numeric type and the incoming <xcFile> data has more digits than the current field length, a runtime error occurs.

Examples

- This example demonstrates an APPEND FROM command using a fields list and a condition:

```
USE Sales NEW
APPEND FROM BranchFile FIELDS Branch, Salesman, Amount;
   FOR Branch = 100
```

- This example demonstrates how a <scope> can be specified to import a particular record from another database file:

```
APPEND RECORD 5 FROM Temp
```

Files: Library is CLIPPER.LIB.

See Also: COPY TO

AVERAGE
Average numeric expressions in the current work area

Syntax

```
AVERAGE <nExp list> TO <idVar list>
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
```

Arguments

<nExp list> is a list of the numeric values to AVERAGE for each record processed.

TO <idVar list> identifies a list of receiving variables which will contain the average results. Variables that either do not exist or are not visible are created as private variables. <idVar list> must contain the same number of elements as <nExp list>.

<scope> defines the portion of the current database file to AVERAGE. The default scope is ALL.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to AVERAGE within the given scope.

Description

AVERAGE calculates the average of one or more numeric expressions to variables for a range of records in the current database file. Zero values are counted in the AVERAGE unless explicitly ruled out with a FOR condition.

Examples

. This example averages a single numeric field using a condition to select a subset of records from the database file:

```
USE Sales NEW
AVERAGE Amount TO nAvgAmount FOR Branch = "100"
```

. This example finds the average date for a range of dates:

```
AVERAGE (SaleDate - CTOD("00/00/00")) ;
   TO nAvgDays FOR !EMPTY(SaleDate)
   dAvgDate := CTOD("00/00/00") + nAvgDays
```

Files: Library is CLIPPER.LIB.

See Also: DBEVAL() SUM TOTAL

CALL*
Execute a C or Assembler procedure

Syntax

CALL <idProcedure> [WITH <exp list>]

Arguments

<idProcedure> is the name of the external procedure to CALL.

WITH <exp list> is an optional list of up to seven expressions of any data type to pass to the external procedure.

Description

CALL executes a separately compiled or assembled procedure. The procedure must be defined as FAR and end with a FAR return instruction. Place parameters on the stack using the C parameter passing convention. Each parameter consists of a FAR (four-byte) pointer to the actual parameter value. When necessary you may use the WORD() function to pass a two-byte binary value in the WITH expression. The DX:BX and ES:BX registers also contain a copy of the first four bytes of parameter information.

The procedure must preserve the BP, SS, SI, DI, ES, and DS registers as well as clear the direction flag.

CALL is a compatibility command and therefore not recommended. It is superseded by the Extend System which provides functions for passing data to and from CA-Clipper.

Notes

. **Character strings:** Pass a character argument as a FAR pointer to a null-terminated string (a string with a 00 hex byte at the end).

. **Numeric values:** Pass each numeric argument as a FAR pointer to an eight-byte IEEE floating point value. To pass a parameter as an integer, use the WORD() function. The WORD() function converts the numeric value to a two-byte binary integer, and passes the integer value directly rather than through a pointer. Note that WORD() will not work for values outside of the #32,767 range since these values cannot be accurately represented as two-byte integers.

. **Date values:** Pass each date argument as a FAR pointer to a four-byte (long) integer containing a Julian day number.

. **Logical values:** Pass each logical argument as a FAR pointer to a two-byte binary integer containing either zero or one for true (.T.) or false (.F.), respectively.

. **Compiling and linking:** CALLed programs must conform to the following rules:

  - Procedures must be in INTEL 8086 relocatable object file format with the .OBJ file extension.

  - Procedures must follow the C calling and parameter passing conventions.

  - Procedures must be available to the linker at link time, along with the library of the source compiler. You will need runtime support for any language other than assembly language. See your compiler manual for further information.

. **Microsoft C:** Microsoft C versions 5.0 and above place a

leading underscore on function names when they are compiled. To call them, therefore, you must CALL _<function>.

- **dBASE III PLUS:** To convert a dBASE III PLUS load module to a CA-Clipper-compatible module, add the following statements to your .ASM file:

  ```
  PUBLIC <proc>
  ```

  and

  ```
  push ds
  mov ds, dx
  ```

Warning! Modifying the parameter values may produce incorrect or unexpected results and, therefore, is strongly discouraged.

Files: Library is CLIPPER.LIB.

See Also: WORD()*

CANCEL*
Terminate program processing

Syntax

CANCEL* | QUIT

Description

CANCEL and QUIT both terminate the current program, closing all open files, and returning control to the operating system. You can use either command from anywhere in a CA-Clipper program system. A RETURN executed at the highest level procedure, performs the same action.

Notes

. **Return code:** When a CA-Clipper program terminates, the return code is set to 1 if the process ends with a fatal error. If the process ends normally, the return code is set to 0 or the last ERRORLEVEL() set in the program.

Files: Library is CLIPPER.LIB.

See Also: ERRORLEVEL() QUIT RETURN

CLEAR ALL*
Close files and release public and private variables

Syntax

CLEAR ALL

Description

CLEAR ALL releases all public and private variables, closes all open databases and related files in all active work areas, and SELECTs work area 1. Related files are index, alternate, and memo files. CLEAR ALL, however, does not release local or static variables.

CLEAR ALL is a compatibility command and not recommended. Its usage in CA-Clipper is superseded by the command or function that performs the specific action you need. You can close files associated with work areas with one of the various forms of the CLOSE command. You can release private and public variables using the RELEASE command although explicitly releasing variables is discouraged in most instances. For more information on the scope and lifetime of variables, refer to the Basic Concepts chapter in the Programming and Utilities guide.

Files: Library is CLIPPER.LIB.

See Also: CLEAR GETS CLEAR MEMORY CLOSE RELEASE

CLEAR GETS
Release Get objects from the current GetList array

Syntax

CLEAR GETS

Description

CLEAR GETS explicitly releases all Get objects in the current and visible GetList array, and terminates the calling READ, releasing any remaining objects in the calling READ, if executed within a SET KEY procedure or a user-defined function invoked by a VALID clause. CLEAR GETS releases Get objects by assigning an empty array to the variable GetList. GetList is the name of the variable used to hold an array of Get objects for subsequent READ commands. There are two other mechanisms that automatically release Get objects: CLEAR specified without the SCREEN clause, and READ specified without the SAVE clause.

CLEAR GETS has two basic uses. First, it can be used to terminate a READ from a SET KEY procedure or VALID user-defined function. Second, it can be used to delete Get objects from the GetList array when you haven't executed a READ or you have saved the Get objects by using READ SAVE.

Files: Library is CLIPPER.LIB.

See Also: @...CLEAR @...GET CLOSE READ RELEASE SET TYPEAHEAD

CLEAR MEMORY
Release all public and private variables

Syntax

CLEAR MEMORY

Description

CLEAR MEMORY deletes both public and private memory variables from the memory variable table. It operates in contrast to RELEASE ALL which does not actually delete public and private memory variables, but assigns NIL to those whose scope is the current procedure. CLEAR MEMORY is the only way to delete all public memory variables from current memory. Local and static variables, however, are unaffected by CLEAR MEMORY.

For more information on variables, refer to the Variables section of the Basic Concepts chapter in the Programming and Utilities guide.

Files: Library is CLIPPER.LIB.

See Also: RELEASE

CLEAR SCREEN
Clear the screen and home the cursor

Syntax

CLEAR [SCREEN] | CLS

Arguments

SCREEN suppresses the automatic releasing of Get objects from the current and visible GetList array when the screen is CLEARed.

Description

CLEAR is a full-screen command that erases the screen, releases pending GETs, and positions the cursor at row and column zero. If the SCREEN clause is specified, Get objects are not released.

CLS is a synonym for CLEAR SCREEN.

Notes

. **SET KEY and VALID:** If you are editing GETs, executing a CLEAR within a SET KEY procedure or within a VALID user-defined function will abort the active READ when control returns. To clear the screen without CLEARing GETs, use either the CLEAR SCREEN or CLS commands.

Files: Library is CLIPPER.LIB.

See Also: @...CLEAR CLEAR GETS SCROLL()

CLEAR TYPEAHEAD
Empty the keyboard buffer

Syntax

CLEAR TYPEAHEAD

Description

CLEAR TYPEAHEAD is a keyboard command that clears all pending keys from the CA-Clipper keyboard buffer. This is useful in user interface procedures or user-defined functions to guarantee that keys processed from the keyboard buffer are appropriate to the current activity and not pending from a previous activity. User functions called by ACHOICE() and DBEDIT() are especially sensitive to such keys.

Note that both the SET TYPEAHEAD and KEYBOARD commands also clear the keyboard buffer.

Examples

. This example empties the keyboard buffer before invoking DBEDIT(), guaranteeing that any pending keys will not be executed:

```
CLEAR TYPEAHEAD
DBEDIT()
```

Files: Library is CLIPPER.LIB.

See Also: KEYBOARD SET TYPEAHEAD

ADEL()
Delete an array element

Syntax

ADEL(<aTarget>, <nPosition>) --> aTarget

Arguments

<aTarget> is the array to delete an element from.

<nPosition> is the position of the target array element to delete.

Returns

ADEL() returns a reference to the target array, <aTarget>.

Description

ADEL() is an array function that deletes an element from an array. The contents of the specified array element is lost, and all elements from that position to the end of the array are shifted up one element. The last element in the array becomes NIL.

Warning! CA-Clipper implements multidimensional arrays by nesting arrays within other arrays. If the <aTarget> array is a multidimensional array, ADEL() can delete an entire subarray specified by <nPosition>, causing <aTarget> to describe an array with a different structure than the original.

Examples

. This example creates a constant array of three elements, and then deletes the second element. The third element is moved up one position, and the new third element is assigned a NIL:

```
LOCAL aArray
aArray := { 1, 2, 3 }      // Result: aArray is
                           // now { 1, 2, 3 }
ADEL(aArray, 2)            // Result: aArray is
                           // now { 1, 3, NIL }
```

Files: Library is CLIPPER.LIB.

See Also: ACOPY() AFILL() AINS()

CLOSE
Close a specific set of files

Syntax

```
CLOSE [<idAlias> | ALL | ALTERNATE | DATABASES |
   FORMAT |INDEXES]
```

Arguments

<idAlias> specifies the work area where all files are CLOSEd.

ALL CLOSEs alternate, database, and index files in all work areas, releasing all active filters, relations, and format definitions.

ALTERNATE CLOSEs the currently open alternate file, performing the same action as SET ALTERNATE TO with no argument.

DATABASES CLOSEs all open databases, memo and associated index files in all work areas, and releases all active filters and relations. It does not, however, have any effect on the active format.

FORMAT releases the current format, performing the same action as SET FORMAT TO with no argument.

INDEXES CLOSEs all index files open in the current work area.

Description

CLOSE is a general purpose command that closes various types of CA-Clipper files depending on the specified option. CLOSE with no option closes the current database and associated files, the same as USE with no arguments.

In CA-Clipper, a number of other commands also CLOSE files including:

- QUIT
- CANCEL*
- RETURN from the highest level procedure
- CLEAR ALL*
- USE with no argument

Files: Library is CLIPPER.LIB.

See Also: QUIT RETURN SET ALTERNATE SET INDEX USE

COMMIT
Perform a solid-disk write for all active work areas

Syntax

COMMIT

Description

COMMIT is a database command that flushes CA-Clipper buffers and performs a solid-disk write for all work areas with open database and index files. The solid-disk write capability is available under DOS version 3.3 and above. Under DOS 3.2 or less, COMMIT flushes CA-Clipper buffers to DOS.

In a network environment, issuing a GO TO RECNO() or a SKIP0 will flush CA-Clipper's database and index buffers, but only a COMMIT will flush the buffers and perform a solid-disk write. Thus to insure updates are visible to other processes, you must issue a COMMIT after all database update commands (e.g., APPEND, REPLACE). To insure data integrity, COMMIT should be issued before an UNLOCK operation. Refer to the Network Programming chapter in the Programming and Utilities guide for more information on update visibility.

Notes

. COMMIT uses DOS interrupt 21h function 68h to perform the solid-disk write. It is up to the network operating system to properly implement this request. Check with the network vendor to see if this is supported.

Examples

. In this example, COMMIT forces a write to disk after a series of memory variables are assigned to field variables:

```
USE Sales EXCLUSIVE NEW
MEMVAR->Name := Sales->Name
MEMVAR->Amount := Sales->Amount
//
@ 10, 10 GET MEMVAR->Name
@ 11, 10 GET MEMVAR->Amount
READ
//
IF UPDATED()
   APPEND BLANK
   REPLACE Sales->Name WITH MEMVAR->Name
   REPLACE Sales->Amount WITH MEMVAR->Amount
   COMMIT
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: DBCOMMIT() DBCOMMITALL() GO SKIP

CONTINUE
Resume a pending LOCATE

Syntax

CONTINUE

Description

CONTINUE is a database command that searches from the current record position for the next record meeting the most recent LOCATE condition executed in the current work area. It terminates when a match is found or end of file is encountered. If CONTINUE is successful, the matching record becomes the current record and FOUND() returns true (.T.); if unsuccessful, FOUND() returns false (.F.).

Each work area may have an active LOCATE condition. In CA-Clipper, a LOCATE condition remains pending until a new LOCATE condition is specified. No other commands release the condition.

Notes

- **Scope and WHILE condition:** Note that the scope and WHILE condition of the initial LOCATE are ignored; only the FOR condition is used with CONTINUE. If you are using a LOCATE with a WHILE condition and want to continue the search for a matching record, use SKIP and then repeat the original LOCATE statement adding REST as the scope.

Examples

- This example scans records in Sales.dbf for a particular salesman and displays a running total sales amounts:

```
LOCAL nRunTotal := 0
USE Sales NEW
LOCATE FOR Sales->Salesman = "1002"
DO WHILE FOUND()
   ? Sales->Salesname, nRunTotal += Sales->Amount
   CONTINUE
ENDDO
```

- This example demonstrates how to continue if the pending LOCATE scope contains a WHILE condition:

```
LOCAL nRunTotal := 0
USE Sales INDEX Salesman NEW
SEEK "1002"
LOCATE REST WHILE Sales->Salesman = "1002";
      FOR Sales->Amount > 5000
DO WHILE FOUND()
   ? Sales->Salesname, nRunTotal += Sales->Amount
   SKIP
   LOCATE REST WHILE Sales->Salesman = "1002";
      FOR Sales->Amount > 5000
ENDDO
```

Files: Library is CLIPPER.LIB.

See Also: EOF() FOUND() LOCATE SEEK

COPY FILE
Copy a file to a new file or to a device

Syntax

COPY FILE <xcSourceFile> TO <xcTargetFile>|<xcDevice>

Arguments

<xcSourceFile> is the name of the source file to COPY including the extension.

<xcTargetFile> is the name of the target file including the extension.

Both arguments can be specified as literal filenames or as character expressions enclosed in parentheses. COPY FILE supplies no default extensions.

<xcDevice> is the name of the device where all subsequent output will be sent. You can specify a device name as a literal character string or a character expression enclosed in parentheses. Additionally, a device can be either local or network. If you COPY TO a non-existing device you create a file with the name of the device. When specifying device names, do not use a trailing colon.

Description

COPY FILE is a file command that copies files to and from the CA-Clipper default drive and directory unless you specify a drive and/or path. If the <xcTargetFile> exists, it is overwritten without warning or error.

Examples

. This example copies a file to a new file and then tests for the existence of the new file:

```
COPY FILE Test.prg TO Real.prg
? FILE("Real.prg")                  // Result: .T.
```

Files: Library is EXTEND.LIB.

See Also: COPY TO RENAME SET DEFAULT

COPY STRUCTURE
Copy the current (.dbf) structure to a new (.dbf) file

Syntax

COPY STRUCTURE [FIELDS <idField list>]
TO <xcDatabase>

Arguments

FIELDS <idField list> defines the set of fields to COPY to the new database structure in the order specified. The default is all fields.

TO <xcDatabase> is the name of the target database file and can be specified either as a literal database filename or as a character expression enclosed in parentheses. The default extension is (.dbf) unless another is specified.

Description

COPY STRUCTURE is a database command that creates an empty database file with field definitions from the current database file. If <xcDatabase> exists, it is overwritten.

COPY STRUCTURE creates empty structures that can be used to archive records from the current database file or to create a temporary database file for data entry.

Examples

. In this example, COPY STRUCTURE creates a temporary file. After the user enters data into the temporary file, the master database file is updated with the new information:

```
USE Sales NEW
COPY STRUCTURE TO Temp
USE Temp NEW
lMore := .T.
DO WHILE lMore
   APPEND BLANK
   @ 10, 10 GET Temp->Salesman
   @ 11, 11 GET Temp->Amount
   READ
   IF UPDATED()
      SELECT Sales
      APPEND BLANK
      REPLACE Sales->Salesman WITH Temp->Salesman
      REPLACE Sales->Amount WITH Temp->Amount
      SELECT Temp
      ZAP
   ELSE
      lMore := .F.
   ENDIF
ENDDO
CLOSE DATABASES
```

Files: Library is CLIPPER.LIB.

See Also: COPY STRU EXTE CREATE

COPY STRUCTURE EXTENDED
Copy field definitions to a (.dbf) file

Syntax

```
COPY STRUCTURE EXTENDED
   TO <xcExtendedDatabase>
```

Arguments

TO <xcExtendedDatabase> is the name of the target structure extended database file. This argument can be specified either as a literal database filename or as a character expression enclosed in parentheses.

Description

COPY STRUCTURE EXTENDED creates a database file whose contents is the structure of the current database file with a record for the definition of each field. The structure extended database file has the following structure:

Structure of an Extended File

Field	Name	Type	Length	Decimals
1	Field_name	Character	10	
2	Field_type	Character	1	
3	Field_len	Numeric	3	0
4	Field_dec	Numeric	4	0

Used in application programs, COPY STRUCTURE EXTENDED permits you to create or modify the structure of a database file programmatically. To create a new database file from the structure extended file, use CREATE FROM. If you need an empty structure extended file, use CREATE.

Notes

. **Character field lengths greater than 255:** In CA-Clipper, the maximum character field length is 64K. For compatibility reasons, field lengths greater than 255 are represented as a combination of the Field_dec and Field_len fields. After COPYing STRUCTURE EXTENDED, you can use the following formula to determine the length of any character field:

```
nFieldLen := IF(Field_type = "C" .AND. ;
      Field_dec != 0, Field_dec * 256) + ;
      Field_len, Field_len)
```

Examples

. This example creates Struc.dbf from Sales.dbf as a structure extended file then lists the contents of Struc.dbf to illustrate the typical layout of field definitions:

```
USE Sales NEW
COPY STRUCTURE EXTENDED TO Struc
USE Struc NEW
LIST Field_name, Field_type, Field_len, Field_dec
```

Result:

```
1 BRANCH        C      3      0
2 SALESMAN      C      4      0
3 CUSTOMER      C      4      0
4 PRODUCT       C     25      0
5 AMOUNT        N      8      2
```

```
6 NOTES          C       0    125
// Field length is 32,000 characters
```

Files: Library is CLIPPER.LIB.

See Also: COPY STRUCTURE CREATE CREATE FROM FIELDNAME() TYPE()

COPY TO
Export records to a new (.dbf) or ASCII file

Syntax

COPY [FIELDS <idField list>] TO <xcFile>
[<scope>] [WHILE <lCondition>] [FOR <lCondition>]
[SDF | DELIMITED [WITH BLANK | <xcDelimiter>]]

Arguments

FIELDS <idField list> specifies the list of fields to copy to the target file. The default is all fields.

TO <xcFile> specifies the name of the target file. The filename can be specified either as a literal filename or as a character expression enclosed in parentheses. If SDF or DELIMITED is specified, (.txt) is the default extension. Otherwise, (.dbf) is the default extension.

<scope> defines the portion of the current database file to COPY. The default is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to COPY within the given scope.

SDF specifies the output file type as a System Data Format ASCII text file. Records and fields are fixed length.

DELIMITED specifies the output file type as a delimited ASCII text file where character fields are enclosed in double quote marks (the default delimiter). Records and fields are variable length.

DELIMITED WITH BLANK identifies an ASCII text file in which fields are separated by one space and character fields have no delimiters.

DELIMITED WITH <xcDelimiter> identifies a delimited ASCII text file where character fields are enclosed using the specified delimiter. <xcDelimiter> can be specified either as a literal character or as a character expression enclosed in parentheses.

See the tables below for more information regarding the format specifications for ASCII text files created using these arguments.

Note: If the DELIMITED WITH clause is specified on a COPY or APPEND command, it must be the last clause specified.

Description

COPY TO is a database command that copies all or part of the current database file to a new file. Records contained in the active database file are copied unless limited by a <scope>, a FOR|WHILE clause, or a filter.

If DELETED is OFF, deleted records in the source file are copied to <xcFile> where they retain their deleted status. If DELETED is ON, however, no deleted records are copied. Similarly, if a FILTER has been SET invisible records are not copied.

Records are copied in controlling index order if there is an index open in the current work area and SET ORDER is not zero. Otherwise, records are copied in natural order.

In a network environment, CA-Clipper opens the target database file EXCLUSIVEly before the COPY TO operation begins. Refer to the Network

Programming chapter in the Programming and Utilities guide for more information.

This table shows the format specifications for SDF text files.

SDF Text File Format Specifications

File Element	Format
Character fields	Padded with trailing blanks
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Padded with leading blanks for zeros
Field separator	None
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

This table shows the format specifications for DELIMITED and DELIMITED WITH <xcDelimiter> ASCII text files.

DELIMITED Text File Format Specifications

File Element	Format
Character fields	Delimited, with trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros truncated
Field separator	Comma
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

This table shows the format specifications for DELIMITED WITH BLANK ASCII text files.

DELIMITED WITH BLANK Text File Format Specifications

File Element	Format
Character fields	Not delimited, trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros truncated
Field separator	Single blank space
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

Examples

. This example demonstrates COPYing to another database file:

```
USE Sales NEW
COPY TO Temp
```

. This example demonstrates the layout of an SDF file with four fields, one for each data type:

```
USE Testdata NEW
COPY NEXT 1 TO Temp SDF
TYPE Temp.txt
```

Result:

```
Character     12.0019890801T
```

. This example demonstrates the layout of a DELIMITED file:

```
COPY NEXT 1 TO Temp DELIMITED
TYPE Temp.txt

Result:

"Character",12.00,19890801,T
```

. This example demonstrates the layout of a DELIMITED file WITH a different delimiter:

```
COPY NEXT 1 TO Temp DELIMITED WITH '
TYPE Temp.txt

Result:

'Character',12.00,19890801,T
```

Files: Library is CLIPPER.LIB.

See Also: APPEND FROM COPY FILE COPY STRUCTURE SET DELETED

COUNT
Tally records to a variable

Syntax

```
COUNT TO <idVar>
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
```

Arguments

TO <idVar> identifies the variable that holds the COUNT result. A variable that either does not exist or is invisible is created as a private variable whose scope is the current procedure.

<scope> is the portion of the current database file to COUNT. The default is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to COUNT within the given scope.

Description

COUNT tallies the number of records from the current work area that match the specified record scope and conditions. The result is then placed in the specified variable. <idVar> can be a variable of any storage class including a field.

Examples

. This example demonstrates a COUNT of Branches in Sales.dbf:

```
USE Sales NEW
COUNT TO nBranchCnt FOR Branch = 100
? nBranchCnt                          // Result: 4
```

. This example tallies the number of records in Sales.dbf whose Branch has the value of 100 and assigns the result to the Count field in Branch.dbf for branch 100:

```
USE Branch INDEX Branch NEW
SEEK 100
USE Sales INDEX SalesBranch NEW
SEEK 100
COUNT TO Branch->Count WHILE Branch = 100
```

Files: Library is CLIPPER.LIB.

See Also: AVERAGE DBEVAL() SUM TOTAL

CREATE
Create an empty structure extended (.dbf) file

Syntax

CREATE <xcExtendedDatabase>

Arguments

<xcExtendedDatabase> is the name of the empty structure extended database file. This argument can be specified either as a literal database filename or as a character expression enclosed in parentheses. If no extension is specified, (.dbf) is the default extension.

Description

CREATE produces an empty structure extended database file with the following structure:

Structure of an Extended File

Field	Name	Type	Length	Decimals
1	Field_name	Character	10	
2	Field_type	Character	1	
3	Field_len	Numeric	3	0
4	Field_dec	Numeric	4	0

Like COPY STRUCTURE EXTENDED, CREATE can be used in conjunction with CREATE FROM to form a new database file. Unlike COPY STRUCTURE EXTENDED, CREATE produces an empty database file and does not require the presence of another database file to create it.

<xcExtendedDatabase> is automatically opened in the current work area after it is CREATEd.

Examples

. This example creates a new structure extended file, places the definition of one field into it, and then CREATEs a new database file FROM the extended structure:

```
CREATE TempStru
APPEND BLANK
REPLACE Field_name WITH "Name",;
   Field_type WITH "C",;
   Field_len WITH 25,;
   Field_dec WITH 0
CLOSE
CREATE NewFile FROM TempStru
```

Files: Library is CLIPPER.LIB.

See Also: COPY STRU EXTE CREATE FROM

CREATE FROM
Create a new (.dbf) file from a structure extended file

Syntax

CREATE <xcDatabase> FROM <xcExtendedDatabase> [NEW]
[ALIAS <xcAlias>] [VIA <cDriver>]

Arguments

<xcDatabase> is the name of the new database file to create from the structure extended file.

<xcExtendedDatabase> is the name of a structure extended file to use as the structure definition for the new database file.

Both of these arguments can be specified either as literal database filenames or as character expressions enclosed in parentheses. If an extension is not specified, the default is (.dbf).

NEW opens <xcDatabase> in the next available work area making it the current work area. If this clause is not specified, <xcDatabase> is opened in the current work area.

ALIAS <xcAlias> is the name to associate with the work area when <xcDatabase> is opened. You may specify the alias name as a literal name or as a character expression enclosed in parentheses. A valid <xcAlias> may be any legal identifier (i.e., it must begin with an alphabetic character and may contain numeric or alphabetic characters and the underscore). Within a single application, CA-Clipper will not accept duplicate aliases. If this clause is not specified, the alias defaults to the database filename.

VIA <cDriver> specifies the replaceable database driver (RDD) to use to process the current work area. <cDriver> is name of the RDD specified as a character expression. If you specify <cDriver> as a literal value, you must enclose it in quotes.

Description

CREATE FROM produces a new database file with the field definitions taken from the contents of a structure extended file. To qualify as a structure extended file, a database file must contain the following four fields:

Structure of an Extended File

Field	Name	Type	Length	Decimals
1	Field_name	Character	10	
2	Field_type	Character	1	
3	Field_len	Numeric	3	0
4	Field_dec	Numeric	4	0

<xcDatabase> is automatically opened in the current work area after it is CREATEd.

Notes

. **Data dictionaries:** For data dictionary applications, you can have any number of other fields within the structure extended file to describe the extended field attributes. You may, for example, want to have fields to describe such field attributes as a description, key flag, label, color, picture, and a validation expression for the VALID clause. When you CREATE FROM, CA-Clipper creates the new database file from the required fields only, ignoring all other

fields in the extended structure. Moreover, CA-Clipper is not sensitive to the order of the required fields.

. **Character field lengths greater than 255:** There are two methods for creating a character field with a length greater than 255 digits:

- Specify the field length using both the Field_len and Field_dec fields according to the following formulation:

```
FIELD->Field_len := <nFieldLength> % 256
FIELD->Field_dec := INT(<nFieldLength> / 256)
```

- Modify the structure of the extended structure changing the length of Field_len to 5 then specify the actual field length. This can be accomplished in DBU.EXE. Refer to the Database Utility chapter in the Programming and Utilities guide for more information.

Examples

. This example is a procedure that simulates an interactive CREATE utility:

```
CreateDatabase("NewFile")
RETURN

FUNCTION CreateDatabase( cNewDbf )
   CREATE TmpExt         // Create empty structure extended
   USE TmpExt
   lMore := .T.
   DO WHILE lMore        // Input new field definitions
      APPEND BLANK
      CLEAR
      @ 5, 0 SAY "Name.....: " GET Field_name
      @ 6, 0 SAY "Type.....: " GET Field_type
      @ 7, 0 SAY "Length...: " GET Field_len
      @ 8, 0 SAY "Decimals.: " GET Field_dec
      READ
      lMore := (!EMPTY(Field_name))
   ENDDO

   // Remove all blank records
   DELETE ALL FOR EMPTY(Field_name)   PACK
   CLOSE

// Create new database file
   CREATE (cNewDbf) FROM TmpExt
   ERASE TmpExt.dbf
   RETURN NIL
```

. This example creates a new definition in a structure extended file for a character field with a length of 4000 characters:

```
APPEND BLANK
REPLACE Field_name WITH "Notes",;
   Field_type WITH "C",;
   Field_len  WITH 4000 % 256,;
   Field_dec  WITH INT(4000 / 256)
```

Files: Library is CLIPPER.LIB.

See Also: COPY STRUCTURE COPY STRU EXTE CREATE

ADIR()*
Fill a series of arrays with directory information

Syntax

```
ADIR([<cFilespec>],
   [<aFilenames>],
   [<aSizes>],
   [<aDates>],
   [<aTimes>],
   [<aAttributes>]) --> nFiles
```

Arguments

<cFilespec> is the path specification of files to include in the scan of the DEFAULT directory. It is a standard file specification that can include the wildcard characters * and ?, as well as a drive and path reference. If omitted, the default specification is *.*.

<aFilenames> is the array to fill with the filenames matching <cFilespec>. Each element contains the filename and extension as a character string in all uppercase letters.

<aSizes> is the array to fill with the sizes of the corresponding files in the <aFilenames> array. Each element is numeric data type.

<aDates> is the array to fill with the dates of the corresponding files in the <aFilenames> array. Each element is a date data type.

<aTimes> is the array to fill with the times of the corresponding files in the <aFilenames> array. Each element filled contains a character string of the form: hh:mm:ss.

<aAttributes> is the array to fill with attributes of the corresponding files in the <aFilenames> array. Each element is a character string. If <aAttributes> is specified, hidden, system, and directory files are included as well as normal files. If <aAttributes> is not specified only normal files are included.

Returns

ADIR() returns the number of files matching the directory skeleton described in <cFilespec>.

Description

ADIR() is an array function that performs two basic operations. First, it returns the number of files matching the file specification. Second, it fills a series of arrays with filenames, sizes, dates, times, and attributes.

ADIR() is a compatibility function and therefore not recommended. It is superseded by the DIRECTORY() function which returns all file information in a multidimensional array.

Notes

. **Directories:** If you specify the <aAttributes> argument and <cFilespec> is *.*, directories will be included in <aFilenames>. In the <aAttributes> array, directories are indicated with an attribute value of "D." If ADIR() is executed within a subdirectory, the first two entries of the <aFilenames> array are "." and "..", the parent and current directory aliases. The date and time of last update are reported for directories, but the size of a directory is always zero.

Examples

. This example creates an array to hold the names of all (.txt) files in the current DEFAULT directory, then uses AEVAL() to list them to the console:

```
LOCAL aFiles[ADIR("*.TXT")]
ADIR("*.TXT", aFiles)
AEVAL(aFiles, { |element| QOUT(element) })
```

Files: Library is EXTEND.LIB.

See Also: ACHOICE() AEVAL() ASCAN() ASORT() DIRECTORY() LEN()

DELETE
Mark records for deletion

Syntax

DELETE [<scope>] [WHILE <lCondition>]
[FOR <lCondition>]

Arguments

<scope> is the portion of the current database file to DELETE. If a scope is not specified, DELETE acts only on the current record. If a conditional clause is specified, the default becomes ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to DELETE within the given scope.

Description

DELETE is a database command that tags records so they can be filtered with SET DELETED ON, queried with DELETED(), or physically removed from the database file with PACK. In addition, display commands such as LIST and DISPLAY identify deleted records with an asterisk (*) character. Once records are deleted, you can reinstate them by using RECALL. If you want to remove all records from a database file, use ZAP instead of DELETE ALL and PACK.

In a network environment, DELETE requires the current record be locked with RLOCK() if you are DELETEing a single record. If you are DELETEing several records, the current database file must be locked with FLOCK() or USEed EXCLUSIVEly. Refer to the Network Programming chapter in the Programming and Utilities guide for more information.

Notes

- **DELETE with SET DELETED ON:** If the current record is deleted with SET DELETED ON, it will be visible until the record pointer is moved.

Examples

- This example demonstrates use of the FOR clause to mark a set of records for deletion:

```
USE Sales INDEX Salesman NEW
DELETE ALL FOR Inactive
```

Files: Library is CLIPPER.LIB.

See Also: DBEVAL() DELETED() PACK RECALL SET DELETED ZAP

DELETE FILE
Remove a file from disk

Syntax

DELETE FILE | ERASE <xcFile>

Arguments

<xcFile> is the name of the file to be deleted from disk and can be specified either as a literal filename or as a character expression enclosed in parentheses. You must specify the filename, including the extension, and it may optionally be preceded by a drive and/or path specification.

Description

DELETE FILE, a synonym for ERASE, is a file command that removes the specified file from disk. SET DEFAULT and SET PATH do not affect DELETE FILE. The file is deleted from disk only if found in the current DOS directory or in the directory explicitly specified as part of the filename.

Warning! Files must be CLOSEd before deleting them.

Examples

. This example removes a specified file from disk then tests to see if the file was in fact removed:

```
? FILE("Temp.dbf")              // Result: .T.
DELETE FILE Temp.dbf
? FILE("Temp.dbf")              // Result: .F.
```

Files: Library is CLIPPER.LIB.

See Also: CLOSE CURDIR() FILE() USE

DIR*
Display a listing of files from a specified path

Syntax

DIR [<xcFileSpec>]

Arguments

<xcFileSpec> is a template that must be matched by displayed files and can include standard wildcard (* and ?) characters. You can specify <xcFileSpec> as a literal file specification string or as a character expression enclosed in parentheses.

Description

DIR is a file command that displays a listing of files from the specified path in one of two formats depending on whether you specify the <xcFileSpec> argument. If you do not specify a path, DIR displays a standard listing of database files from the current or specified directory. The list includes the database filename, date of last update, and number of records. Including the option <xcFileSpec> displays a list of all files in the specified directory. This list includes the filename, extension, number of bytes, and date of last update.

If no path is specified, DIR displays files from the current DOS drive and directory unless SET DEFAULT has been used to specify a new default directory.

DIR is a compatibility command and therefore not recommended. It is superseded by the DIRECTORY() function which returns an array of file information from a wildcard pattern. Using this array, you can display file information to screen.

Notes

. **Directory picklists:** To present directory information to the user, use ADIR() to return one or more arrays of directory information, and ACHOICE() to present a picklist to the user.

Examples

. These examples display DOS and database files in the current directory:

```
cFilespec := "*.*"
DIR (cFilespec)            // Display all files
DIR                        // Display all (.dbf) files
DIR *.prg                  // Display all program files
```

Files: Library is EXTEND.LIB.

See Also: DIRECTORY() SET DEFAULT

DISPLAY
Display records to the console

Syntax

```
DISPLAY <exp list>
   [TO PRINTER] [TO FILE <xcFile>]
   [<scope>] [WHILE <lCondition>]
   [FOR <lCondition>] [OFF]
```

Arguments

<exp list> is the list of values to display for each record processed.

TO PRINTER echoes output to the printer.

TO FILE <xcFile> echoes output to the indicated file which can be specified either as a literal filename or as a character expression enclosed in parentheses. If an extension is not specified, (.txt) is added.

<scope> is the portion of the current database file to DISPLAY. The default is the current record, or NEXT 1. If a condition is specified, the scope becomes ALL.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to DISPLAY within the given scope.

OFF suppresses the display of the record number.

Description

DISPLAY is a database command that sequentially accesses records in the current work area sending the results of the <exp list> to the console in a tabular format with each column separated by a space. The command does not display column headers or pause at predetermined intervals. DISPLAY is identical to LIST with the exception that its default scope is NEXT 1 rather than ALL.

When invoked, output is sent to the screen and optionally to the printer and/or a file. To suppress output to the screen while printing or echoing output to a file, SET CONSOLE OFF before the DISPLAY command line.

Notes

- **Interrupting output:** To let the user interrupt the processing of a DISPLAY command, using the INKEY() function, add a test for the interrupt key press to the FOR condition. See the example below.

- **Printer margin:** Since DISPLAY is a console command, it honors the current SET MARGIN for output echoed to the printer.

Examples

- This example illustrates a simple DISPLAY, and a conditional DISPLAY to the printer:

```
USE Sales NEW
DISPLAY DATE(), TIME(), Branch
DISPLAY Branch, Salesman FOR Amount > 500 TO PRINTER
```

- This example interrupts a DISPLAY using INKEY() to test

whether the user pressed the Esc key:

```
#define K_ESC  27
USE Sales INDEX SalesMan NEW
DISPLAY Branch, Salesman, Amount WHILE ;
   INKEY() != K_ESC
```

Files: Library is CLIPPER.LIB.

See Also: DBEVAL() INKEY() LIST SET MARGIN

EJECT
Advance the printhead to top of form

Syntax

EJECT

Description

EJECT is a printing command that sends a formfeed character (CHR(12)) to the printer and sets the PCOL() and PROW() values to zero. If you address a printer row less than the last row position since an EJECT or SETPRC() was executed, CA-Clipper automatically performs an EJECT. Because of this, your printing logic must proceed sequentially from left to right down the page. If you need to reset the internal printer row and column values to zero without sending a formfeed, use SETPRC().

Examples

. This example prints a simple list report and uses EJECT to advance to a new page when the line counter reaches the maximum number of lines to print per page:

```
LOCAL nLine := 99, nPage := 0
USE Sales NEW
SET PRINTER ON
SET CONSOLE OFF
DO WHILE !EOF()
   IF nLine > 55
      EJECT
      ? "Page " + LTRIM(STR(++nPage, 3))
      ? "Date " + CTOD(DATE())
      ?
      ? "Salesman", "Amount"
      ?
      nLine := 6
   ENDIF
   ? Sales->Salesman, Sales->Amount
   nLine++
   SKIP
ENDDO
SET PRINTER OFF
SET CONSOLE ON
CLOSE
```

Files: Library is CLIPPER.LIB.

See Also: SET CONSOLE SET DEVICE SET PRINTER SETPRC()

ERASE
Remove a file from disk

Syntax

ERASE | DELETE FILE <xcFile>

Arguments

<xcFile> is the name of the file to be deleted from disk and can be specified either as a literal filename or as a character expression enclosed in parentheses. The filename must be fully specified including the extension, and it may optionally be preceded by a drive and/or path specification.

Description

ERASE is a file command that removes a specified file from disk. SET DEFAULT and SET PATH do not affect ERASE. The file is deleted from disk only if found in the current DOS directory or in the directory explicitly specified as part of the filename.

Warning! Files must be CLOSEd before being ERASEd.

Examples

. This example removes a specified file from disk then tests to see if the file was in fact removed:

```
? FILE("Temp.dbf")         // Result: .T.
ERASE Temp.dbf
? FILE("Temp.dbf")         // Result: .F.
```

Files: Library is CLIPPER.LIB.

See Also: CLOSE CURDIR() FERASE() FILE() FRENAME() RENAME USE

FIND*
Search an index for a specified key value

Syntax

FIND <xcSearchString>

Arguments

<xcSearchString> is part or all of the index key of a record to search for and can be specified either as a literal string or as a character expression enclosed in parentheses. If an expression is specified instead of a literal string, FIND operates the same as SEEK.

Description

FIND is a database command that searches an index for the first key matching the specified character string and positions the record pointer to the corresponding record.

If SOFTSEEK is OFF and FIND does not find a record, the record pointer is positioned to LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.).

If SOFTSEEK is ON, the record pointer is positioned to the record with the first key value greater than the search argument and FOUND() returns false (.F.). In this case, EOF() returns true (.T.) only if there are no keys in the index greater than the search argument.

FIND is a compatibility command and not recommended. Its usage is superseded entirely by the SEEK command.

Examples

. These examples show simple FIND results:

```
USE Sales INDEX Branch NEW
FIND ("500")
? FOUND(), EOF(), RECNO()         // Result: .F. .T. 85
FIND 200
? FOUND(), EOF(), RECNO()         // Result: .T. .F. 5
FIND "100"
? FOUND(), EOF(), RECNO()         // Result: .T. .F. 1
```

Files: Library is CLIPPER.LIB.

See Also: EOF() FOUND() SEEK SET ORDER SET SOFTSEEK

GO
Move the record pointer to a specific record

Syntax

GO[TO] <nRecord> | BOTTOM | TOP

Arguments

<nRecord> specifies the target record number.

BOTTOM specifies the last logical record in the current work area.

TOP specifies the first logical record in the current work area.

Description

GO is a database command that positions the record pointer in the current work area to a specified record. You can specify the record by number or as top or bottom of file. If the new position is TOP and there is an active index, the record pointer moves to the first record in the index, otherwise it moves to record 1. If the new position is BOTTOM and there is an active index, the record pointer moves to the last record in the index, otherwise it moves to LASTREC().

The GO <nRecord> form of the command is the same as the RECORD clause of any database command that supports a scope.

Notes

- **Accessing filtered records:** You can access records filtered by SET DELETED ON or SET FILTER TO with GO <nRecord>. If DELETED is ON or there is an active FILTER, GO BOTTOM moves to the last logical record that is not deleted and/or that matches the filter condition. If DELETED is ON or there is an active FILTER, GO TOP moves to the first logical record that is not deleted and/or that matches the filter condition.

- **Unique indexes:** You can access records not present in an index created with SET UNIQUE ON or INDEX...UNIQUE with GO <nRecord>.

- **Record out of range:** In CA-Clipper, a GO that positions the pointer to a record out of range of the database file does not generate a runtime error. Instead, EOF() and BOF() are set to true (.T.) and the record pointer is set to LASTREC() + 1.

- **Refreshing database and index buffers:** In a network environment, you can refresh database and index buffers without moving the record pointer by using GO RECNO(). Refer to the Network Programming chapter in the Programming and Utilities guide for more information.

Examples

- These examples show simple GO results:

```
USE Sales NEW
? LASTREC()            // Result: 84
//
GO TOP
? RECNO()              // Result: 1
//
GO BOTTOM
? RECNO()              // Result: 84
//
GO 5
? RECNO()              // Result: 5
```

```
//
GO 5 + 15
? RECNO()              // Result: 20
```

Files: Library is CLIPPER.LIB.

See Also: LASTREC() RECNO() SET FILTER SET RELATION SKIP

INDEX
Create an index file

Syntax

```
INDEX ON <expKey> TO <xcIndex> [UNIQUE]
   [<scope>]
   [FOR <lCondition>]
   [WHILE <lCondition>]
   [[EVAL <lCondition>][EVERY <nRecords>]]
   [ASCENDING | DESCENDING]
```

Arguments

<expKey> is an expression that returns the key value to place in the index for each record in the current work area. <expKey> can be character, date, logical, or numeric type. The maximum length of the index key expression is 250 characters.

TO <xcIndex> specifies the name of the index file to create. You can specify the filename either as a literal filename or as a character expression enclosed in parentheses. If the specified name does not include a file extension, a default extension is used. For standard CA-Clipper indexes the default extension is (.ntx). For dBASE III PLUS compatible indexes, the default extension is (.ndx).

UNIQUE specifies that <xcIndex> includes only unique key values.

<scope> is the portion of the current database to INDEX. The default is ALL records. All other valid scope clauses (i.e., NEXT <nRecords>, REST, and RECORD <nRecord>) are permitted. If you specify a scope, the database is processed in the controlling index order. The scope is transient (i.e., is not stored in the (.ntx) file and not used for REINDEXing purposes).

FOR <lCondition> specifies the conditional set of records on which to index. Only those records that meet the condition are included in the resulting index (.ntx) file. The maximum length of the FOR expression is 250 characters. The FOR condition is stored as part of the (.ntx) file and therefore used when updating or recreating the index using the REINDEX command.

WHILE <lCondition> specifies another condition that each record processed must meet; as soon as a record is encountered that causes the condition to fail, the INDEX command terminates. If a WHILE clause is specified, the database file is processed in the controlling index order. The WHILE condition is transient (i.e., is not stored in the (.ntx) file and not used for index updates and REINDEXing purposes).

EVAL <lCondition> specifies a condition that is evaluated either for each record processed or at the interval specified by the EVERY clause. This clause provides a flexible way to monitor the progress of the indexing operation. <lCondition> may be any logical expression, including a user-defined function that displays status information and then returns a logical value. The value of <lCondition> is transient which means that it is discarded after the index is created (i.e., it is not retained for REINDEXing purposes). Indexing stops if the EVAL condition returns false (.F.)

EVERY <nRecords> specifies a numeric expression that modifies how often EVAL is evaluated. When using EVAL, the EVERY option offers a performance enhancement by evaluating the condition for every nth record instead of for each record indexed. The EVERY keyword is ignored if no EVAL condition is specified.

Note: Since <scope>, WHILE, EVAL, and EVERY are transient, they will not be available when the database is REINDEXed. Although the

REINDEX command does provide for EVAL and EVERY clauses, <scope> and WHILE are not allowed as part of the REINDEX operation. Thus, scoping is useful only for immediate queries.

ASCENDING specifies that the index keys be sorted in increasing order of value. If neither ASCENDING nor DESCENDING is specified, ASCENDING is assumed. Although not stored as an explicit part of the (.ntx) file, ASCENDING is an implicit (.ntx) file attribute that is understood by the REINDEX command.

DESCENDING specifies that the index keys be sorted in decreasing order of value. Using this keyword is the same as specifying the DESCEND() function within <expKey>, but without the performance penalty during index updates. If you create a DESCENDING index, you won't need to use the DESCEND() function during a SEEK. DESCENDING is an attribute of the (.ntx) file, where it is stored and used for REINDEXing purposes.

Description

INDEX is a database command that creates a file containing an index of records in the current database file, based on <expKey>. When the index file is used, the database file records appear in key expression order although the index does not alter the physical order of records within the database file. INDEX orders character keys in accordance with the ASCII value of each character within the string, numeric values in numeric order, date values chronological order with blank dates treated as low values, and logical values sorted with true (.T.) treated as high values. Memo fields cannot be INDEXed.

When INDEX ON is invoked, all open index files in the current work area are closed and the new index file is created. When the indexing operation finishes, the new index remains open becoming the controlling index and the record pointer is positioned to the first record in the index.

INDEX is like SORT but does not make a physical copy of the database file and is updated each time a new key value is entered in the current database file. SORT is primarily used to copy an ordered subset of records to another database file.

In a network environment, INDEX opens <xcIndex> exclusive. Failure to open the file raises , a runtime error. Refer to the Network Programming chapter in the Programming and Utilities guide for more information.

Notes

. **Date indexes:** CA-Clipper supports date indexes for both (.ntx) and (.ndx) index types. For a key expression that includes a date as a subset of the key, create the expression as character data type and use DTOS() to convert the date to character.

. **Declared variables:** You cannot use local and static variables in index key expressions. This is because index expressions are saved as text in the index files and are later compiled and run using the macro operator (&) to produce the actual key values. Since locals and statics are invisible within macro variables and expressions, references to them within index key expressions are not resolved properly.

  For similar reasons, a variable declared with any of the other compile-time declarations such as MEMVAR or FIELD, are not valid within an index key expression. If you need aliasing or variable qualification in an index key expression, implement it explicitly in the expression rather than through compiler declarations. Refer to the discussion on variables in the Basic Concepts chapter of the Programming and Utilities guide.

- **Deleted and filtered records:** Records that are filtered or marked for deletion are included in the index.

- **Dictionary order indexes:** To create a dictionary order index of character keys, use UPPER(<expKey>) as the index expression. Dictionary order is an order where capitalization is ignored.

- **Compatible index files:** CA-Clipper supports dBASE III PLUS compatible index files using a special database driver. Refer to the CA-Clipper Linker chapter in the Programming and Utilities guide for instructions on linking drivers.

Warning! The dBASE III PLUS database driver supports the CA-Clipper locking mechanism in place of the dBASE III PLUS mechanism. This means you cannot have both a CA-Clipper program and dBASE III PLUS concurrently accessing the same index files on a network without compromising integrity. With two CA-Clipper programs, however, the dBASE III PLUS database driver works the same as the standard CA-Clipper database driver.

- **Unique indexes:** When you INDEX...UNIQUE or SET UNIQUE ON, CA-Clipper creates an index with uniqueness as an attribute. As indexing proceeds and two or more records have the same key value, CA-Clipper includes only the first record in the index. Whenever the unique index is updated, REINDEXed, or PACKed, only unique records are added. Note that a unique index retains the uniqueness attribute and is unaffected by subsequent invocations of SET UNIQUE.

- **Trimming key values:** In CA-Clipper, an <expKey> that changes the key length may create an inoperable index. This happens because CA-Clipper stores index key values in fixed increments and index key sizes are calculated by evaluating <expKey> on a blank record. A trimmed <expKey>, therefore, always evaluates to a null string ("") leading to a size mismatch between the target and the defined key length. To use any of the TRIM() functions within a key expression, use PADR() to guarantee a fixed length like this:

```
USE Customer NEW
INDEX ON PADR(RTRIM(Last) + First, 40) TO CustName
```

Examples

- This example creates a simple index:

```
USE Customer NEW
INDEX ON Name TO Customer
```

- This example creates a unique index:

```
USE Customer NEW
INDEX ON Name TO Customer UNIQUE
```

- This example creates a simple descending index:

```
USE Customer NEW
INDEX ON Name TO Customer DESCENDING
```

- This example creates an index with a progress indicator which displays the percentage of the database indexed every 10 records:

```
USE Customer NEW
INDEX ON Name TO Customer EVAL NtxProgress();
         EVERY LASTREC()/10

FUNCTION NtxProgress
LOCAL cComplete := LTRIM(STR((RECNO()/LASTREC()) * 100))
@ 23, 00 SAY "Indexing..." + cComplete + "%"
RETURN .T.
```

. This example creates a simple conditional index using a FOR condition:

```
USE Customer NEW
INDEX ON Name TO Customer FOR PostalCode = "95605"
```

. This example creates a subindex using an existing order:

```
USE Customer INDEX Customer NEW
SEEK "SMITH"
IF FOUND()
   INDEX ON PostalCode TO PostCode WHILE Name = "SMITH"
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: DESCEND() DTOS() REINDEX SET INDEX SET ORDER

INPUT
Enter the result of an expression into a variable

Syntax

INPUT [<expPrompt>] TO <idVar>

Arguments

<expPrompt> is an optional prompt displayed before the input area. The prompt can be an expression of any data type.

TO <idVar> specifies the name of the variable to assign the evaluation result. If <idVar> is not visible or does not exist, a private variable is created and assigned the result.

Description

INPUT is a console command and wait state that takes entry from the keyboard in the form of a valid expression of up to 255 characters and any data type. When invoked, INPUT sends a carriage return/linefeed to the screen, displays the prompt, and begins accepting keyboard input at the first character position following the prompt. Input greater than MAXCOL() wraps to the next line.

Pressing Return terminates entry of the expression. The expression is then compiled and evaluated using the macro operator (&), and the result assigned to <idVar>. If the variable is not visible or does not exist, it is created as a private. If no expression is entered, no action is taken.

INPUT supports only two special keys: Backspace and Return. Esc is not supported. Backspace deletes the last character typed. Return confirms entry and is the only key that can terminate an INPUT.

Examples

. In this example INPUT assigns a keyboard entry to an existing local variable:

```
LOCAL exp
INPUT "Expression: " TO exp
IF exp != NIL
   ? exp
ELSE
   ? "No expression entered"
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: ACCEPT

JOIN
Create a new database file by merging records/fields from two work areas

---

Syntax

```
JOIN WITH <xcAlias> TO <xcDatabase>
   FOR <lCondition> [FIELDS <idField list>]
```

Arguments

WITH <xcAlias> is the name of the work area to merge with records from the current work area. You can specify it either as a literal alias or as a character expression enclosed in parentheses.

TO <xcDatabase> is the name of the target database file specified either as a literal filename or as a character expression enclosed in parentheses.

FOR <lCondition> selects only records meeting the specified condition.

FIELDS <idField list> is the projection of fields from both work areas into the new database file. To specify any fields in the secondary work area, reference them with the alias. If the FIELDS clause is not specified, all fields from the primary work area are included in the target database file.

Description

JOIN creates a new database file by merging selected records and fields from two work areas based on a general condition. JOIN works by making a complete pass through the secondary work area for each record in the primary work area, evaluating the condition for each record in the secondary work area. When the <lCondition> is true (.T.), a new record is created in the target database file using the FIELDS specified from both work areas.

If SET DELETED is OFF, deleted records in both source files (i.e., the two files being JOINed) are processed. However, their deleted status is not retained in the target <xcDatabase>. No record in the target file is marked for deletion regardless of its deleted status in either of the source files.

If SET DELETED is ON, no deleted records are processed in either of the source files. Thus, deleted records do not become part of the target <xcDatabase>. Similarly, filtered records are not processed and do not become part of the target file.

Warning! The number of records processed will be the LASTREC() of the primary work area multiplied by the LASTREC() of the secondary work area. For example, if you have two database files with 100 records each, the number of records JOIN processes is the equivalent of sequentially processing a single database file of 10,000 records. Therefore, use this command carefully.

Examples

. This example JOINs the Customer.dbf to the Invoices.dbf to produce Purchases.dbf:

```
USE Invoices NEW
USE Customers NEW
JOIN WITH Invoices TO Purchases;
   FOR Last = Invoices->Last;
   FIELDS First, Last, Invoices->Number, ;
      Invoices->Amount
```

Files: Library is CLIPPER.LIB.

See Also: SET RELATION

KEYBOARD
Stuff a string into the keyboard buffer

Syntax

KEYBOARD <cString>

Arguments

<cString> is the string to stuff into the keyboard buffer.

Description

KEYBOARD is a keyboard command that stuffs <cString> into the keyboard buffer after clearing any pending keystrokes. The characters remain in the keyboard buffer until fetched by a wait state such as ACCEPT, INPUT, READ, ACHOICE(), MEMOEDIT(), and DBEDIT(), or INKEY(). In addition to display characters, <cString> may include control characters.

Generally, to convert an INKEY() code of a control key to a character value, use CHR(). There are, however, some limitations. You can only stuff characters with INKEY() codes between zero and 255, inclusive, into the keyboard buffer.

Typically, KEYBOARD is used in a SET KEY procedure to reassign keys in a wait state. Another use within the ACHOICE() user function is to input the keys you want ACHOICE() to execute before returning control to it. The same concept applies to DBEDIT().

Examples

. This example uses KEYBOARD to stuff a GET with an item selected from a picklist each time the GET is entered:

```
#include "Inkey.ch"
LOCAL cVar1 := SPACE(10), nVar := 2500, ;
      cVar2 := SPACE(10)
CLEAR
@ 09, 10 GET cVar1
@ 10, 10 GET cVar2 WHEN PickList()
@ 11, 10 GET nVar
READ
RETURN

FUNCTION PickList
   STATIC aList := { "First", "Second", ;
      "Three", "Four" }
   LOCAL cScreen, nChoice, nKey := LASTKEY()
   cScreen := SAVESCREEN(10, 10, 14, 20)
   @ 10, 10 TO 14, 20 DOUBLE
   IF (nChoice := ACHOICE(11, 11, 13, 19, aList)) != 0
      KEYBOARD CHR(K_CTL_Y) + aList[nChoice] + ;
            CHR(nKey)

   ENDIF
   RESTSCREEN(10, 10, 14, 20, cScreen)
   RETURN .T.
```

Files: Library is CLIPPER.LIB, header file is Inkey.ch.

See Also: CHR() INKEY() LASTKEY() NEXTKEY() SET KEY

LABEL FORM
Display labels to the console

Syntax

```
LABEL FORM <xcLabel>
   [TO PRINTER] [TO FILE <xcFile>] [NOCONSOLE]
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
   [SAMPLE]
```

Arguments

<xcLabel> is the name of the label (.lbl) file that contains the FORM definition of the LABEL and can be specified either as a literal filename or as a character expression enclosed in parentheses. If an extension is not specified (.lbl) is assumed.

TO PRINTER echoes output to the printer.

TO FILE <xcFile> echoes output to <xcFile>. Specify <xcFile> as a literal filename or as a character expression enclosed in parentheses. If an extension is not specified, (.txt) is added.

NOCONSOLE suppresses all LABEL FORM output to the console. If not specified, output automatically displays to the console unless SET CONSOLE is OFF.

<scope> is the portion of the current database file to display labels. The default is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies to LABEL FORM, the conditional set of records within the given scope.

SAMPLE displays test labels as rows of asterisks. Each test label has the same number of columns and rows as the label definition. Then, following each test label display, is the query, "Do you want more samples?" Answering "Y" forces another test label display. Answering "N" causes LABEL FORM to display the actual labels for the specified scope and condition.

Description

LABEL FORM is a console command that sequentially accesses records in the current work area, displaying labels using a definition stored in a (.lbl) file. Create the (.lbl) using RL.EXE or by dBASE III PLUS. Refer to the Report and Label Utility chapter in the Programming and Utilities guide for more information about creating label definitions.

When invoked, output is sent to the screen, and optionally, to the printer and/or a file. To suppress output to the screen while printing or echoing output to a file, SET CONSOLE OFF before invocation of LABEL FORM.

When invoked, LABEL FORM searches the current SET PATH drive and directory, if the <xcLabel> file is not found in the current directory, and the path is not specified.

Notes

. **Interrupting LABEL FORM:** To interrupt a LABEL FORM, use INKEY() as a part of the FOR condition, to test for an interrupt key press. See the example below.

. **Printer margin:** LABEL FORM obeys the current SET MARGIN for

output echoed to the printer.

Examples

. This example prints a set of labels and writes them to a file with a single command. Two forms of the command are shown:

```
LOCAL cLabel := "Sales", cFile := "Sales"
USE Sales INDEX Sales NEW
LABEL FORM Sales TO PRINTER TO FILE Sales
LABEL FORM (cLabel) TO PRINTER TO FILE (cFile)
```

. This example interrupts LABEL FORM using INKEY() to test if the user pressed the Esc key:

```
#define K_ESC  27
USE Sales INDEX Sales NEW
LABEL FORM Sales WHILE INKEY() != K_ESC
```

Files: Library is CLIPPER.LIB.

See Also: REPORT FORM SET PRINTER

LIST
List records to the console

Syntax

```
LIST <exp list>
   [TO PRINTER] [TO FILE <xcFile>]
   [<scope>] [WHILE <lCondition>]
   [FOR <lCondition>] [OFF]
```

Arguments

<exp list> is the list of expressions to be evaluated and displayed for each record processed.

TO PRINTER echoes output to the printer.

TO FILE <xcFile> echoes output to the specified filename and can be specified either as a literal filename or as a character expression enclosed in parentheses. If an extension is not specified, (.txt) is added.

<scope> is the portion of the current database file to LIST. The default is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to LIST within the given scope.

OFF suppresses the display of record numbers.

Description

LIST is a console command that sequentially accesses records in the current work area, displaying the results of one or more expressions for each record accessed. The output is in tabular format with each column separated by a space. LIST is identical to DISPLAY with the exception that its default scope is ALL rather than NEXT 1.

When invoked, output is sent to the screen and, optionally, to the printer and/or a file. To suppress output to the screen while printing or echoing output to a file, SET CONSOLE OFF before the LIST invocation.

Notes

. **Interrupting LIST:** So the user may interrupt a LIST, use INKEY() as part of the FOR condition to test for an interrupt key press. See the example below.

. **Printer margin:** LIST honors the current SET MARGIN for output echoed to the printer.

Examples

. In this example, a simple list is followed by a conditional list to the printer:

```
USE Sales
LIST DATE(), TIME(), Branch
LIST Branch, Salesman FOR Amount > 500 TO PRINTER
```

. This example interrupts LIST using INKEY() to test whether the user pressed the Esc key:

```
#define K_ESC     27
```

```
USE Sales INDEX Salesman NEW
LIST Branch, Salesman, Amount WHILE INKEY() != K_ESC
```

Files: Library is CLIPPER.LIB.

See Also: ?|?? DISPLAY

LOCATE
Search sequentially for a record matching a condition

Syntax

LOCATE [<scope>] FOR <lCondition>
[WHILE <lCondition>]

Arguments

<scope> is the portion of the current database file in which to perform the LOCATE. The default scope is ALL records.

FOR <lCondition> specifies the next record to LOCATE within the given scope.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

Description

LOCATE is a database command that searches for the first record in the current work area that matches the specified conditions and scope. When you first execute a LOCATE, it searches from the beginning record of the scope for the first matching record in the current work area. It terminates when a match is found or the end of the LOCATE scope is reached. If it is successful, the matching record becomes the current record and FOUND() returns true (.T.). If it is unsuccessful, FOUND() returns false (.F.) and the positioning of the record pointer depends on the controlling scope of the LOCATE.

Each work area can have its own LOCATE condition. The condition remains active until you execute another LOCATE command in that work area or the application terminates.

LOCATE works with CONTINUE. Once a LOCATE has been issued, you can resume the search from the current record pointer position with CONTINUE. There are, however, some exceptions. See note below.

Notes

. **CONTINUE:** Both the <scope> and the WHILE condition apply only to the initial LOCATE and are not operational for any subsequent CONTINUE commands. To continue a pending LOCATE with a scope or WHILE condition, use SKIP then LOCATE REST WHILE <lCondition> instead of CONTINUE.

Examples

. These examples show typical LOCATEs:

```
USE Sales INDEX Salesman
LOCATE FOR Branch = "200"
? FOUND(), EOF(), RECNO()         // Result: .T. .F. 5
LOCATE FOR Branch = "5000"
? FOUND(), EOF(), RECNO()         // Result: .F. .T. 85
```

. This example shows a LOCATE with a WHILE condition that is continued by using LOCATE REST:

```
SEEK "Bill"
LOCATE FOR Branch = "200" WHILE Salesman = "Bill"
DO WHILE FOUND()
   ? Branch, Salesman
   SKIP
   LOCATE REST FOR Branch = "200" WHILE ;
            Salesman = "Bill"
```

```
ENDDO
```

Files: Library is CLIPPER.LIB.

See Also: CONTINUE EOF() FOUND() SEEK SET FILTER

MENU TO
Execute a lightbar menu for defined PROMPTs

Syntax

MENU TO <idVar>

Arguments

<idVar> is the name of the variable to assign the result of the menu selection. If the specified variable is not visible or does not exist, a private variable is created and assigned the result.

Description

MENU TO is the selection mechanism for the CA-Clipper lightbar menu system. Before invoking MENU TO, first define the menu items and associated MESSAGEs with a series of @...PROMPT commands. Then, activate the menu with MENU TO <idVar>. If <idVar> does not exist or is not visible, MENU TO creates it as a private variable and places the highlight on the first menu item. If <idVar> does exist, its initial value determines the first menu item highlighted.

Notes

. **Color:** Menu items are painted in the current standard color with the highlighted menu item appearing in the current enhanced color.

. **Navigation and selection:** Pressing the arrow keys moves the highlight to the next or previous menu item. As each menu item is highlighted the associated MESSAGE displays on the row specified with SET MESSAGE. If WRAP is ON, an Uparrow from the first menu item moves the highlight to the last menu item. Also, a Dnarrow from the last menu item moves the highlight to the first.

  To make a selection, press Return or the first character of a menu item. MENU TO then returns the position of the selected menu item as a numeric value into the specified memory variable. Pressing Esc aborts the menu selection and returns zero. The table below summarizes the active keys within MENU TO.

. **SET KEY procedures:** A MENU TO command can be nested within a SET KEY procedure invoked within a menu without clearing the pending PROMPTs.

  MENU TO Active Keys

Key	Action
Uparrow	Move to previous menu item
Dnarrow	Move to next menu item
Home	Move to first menu item
End	Move to last item menu item
Leftarrow	Move to previous menu item
Rightarrow	Move to next menu item
PgUp	Select menu item, returning position
PgDn	Select menu item, returning position
Return	Select menu item, returning position
Esc	Abort selection, returning zero
First letter	Select first menu item with same first letter, returning position

Examples

. This example creates a simple vertical lightbar menu with

messages appearing centered on line 23. When invoked, the highlight defaults to the second menu item based on the initial value of nChoice:

```
LOCAL nChoice := 2
SET WRAP ON
SET MESSAGE TO 23 CENTER
@ 6, 10 PROMPT "Add"  MESSAGE "New Acct"
@ 7, 10 PROMPT "Edit" MESSAGE "Change Acct"
@ 9, 10 PROMPT "Quit" MESSAGE "Return to DOS"
MENU TO nChoice
//
DO CASE
CASE nChoice = 0
   QUIT
CASE nChoice = 1
   NewAccount()
CASE nChoice = 2
   ChangeAccount()
CASE nChoice = 3
   QUIT
ENDCASE
RETURN
```

Files: Library is CLIPPER.LIB.

See Also: @...PROMPT ACHOICE() SET MESSAGE SET WRAP

NOTE*
Place a single-line comment in a program file

Syntax

NOTE [<commentText>]

Arguments

<commentText> is a string of characters placed after the comment indicator.

Description

NOTE is command synonym for the single-line comment indicator (*). All characters after NOTE are ignored until the CA-Clipper compiler encounters an end of line (carriage return/linefeed). This means a single-line comment cannot be continued with the semicolon onto a new line.

If you need a multiline comment, begin the comment block with a slash-asterisk (/*) symbol and end the comment block with an asterisk-slash (*/) symbol. If you need to comment out a block of code, use the conditional compilation directives #ifdef...#endif instead of multiline comments. This is important since nested comments are illegal.

NOTE is a compatibility command and therefore not recommended. It is superseded by the C-style comment symbols slash-asterisk (/*) and asterisk-slash (*/), as well as the double-slash (//). It is also superseded by the dBASE-style comment symbols, asterisk (*) and the double-ampersand (&&).

For a complete discussion on comment indicators, refer to the Basic Concepts chapter in the Programming and Utilities guide.

Examples

. These examples show the various comment symbols supported by CA-Clipper:

```
// This is a comment
/* This is a comment */
* This is a comment
&& This is a comment
NOTE This is a comment
```

PACK
Remove deleted records from a database file

Syntax

PACK

Description

PACK is a database command that removes all records marked for deletion from the current database file, REINDEXes all active indexes in the current work area, and recovers all the physical space occupied by the deleted records. During its operation, PACK does not create any backup files, although the associated REINDEX operation may. After the PACK command terminates, the record pointer is reset to the first logical record in the current work area.

In a network environment, PACK requires the current database be USEd EXCLUSIVEly. If this condition is not met when PACK is invoked, CA-Clipper generates a runtime error.

Note that PACKing large database files can be a time-consuming process and may not be feasible in a high-volume transaction system on a network. By modifying the system design, you can remove the necessity of physically removing records from the database file altogether. See the Network Programming chapter in the Programming and Utilities guide for more information.

Examples

. The following shows the result of a simple PACK:

```
USE Sales NEW
? LASTREC()                  // Result: 84
//
DELETE RECORD 4
PACK
? LASTREC()                  // Result: 83
```

Files: Library is CLIPPER.LIB.

See Also: DELETE DELETED() FLOCK() RECALL REINDEX ZAP

QUIT
Terminate program processing

Syntax

QUIT | CANCEL*

Description

QUIT and CANCEL both terminate program processing, close all open files, and return control to the operating system. Each of these commands can be used from anywhere in a program system. A RETURN executed at the highest level procedure or a BREAK, with no pending SEQUENCE, also QUITs the program.

Notes

. **Return code:** When a CA-Clipper program terminates, the return code is set to 1 if the process ends with a fatal error. If the process ends normally the return code is set to zero or the last ERRORLEVEL() set in the program.

Examples

. This example uses QUIT in a dialog box:

```
IF DialogYesNo(10, 10, "Quit to DOS", "BG+/B,B/W", 2)
   QUIT
ENDIF
RETURN
```

Files: Library is CLIPPER.LIB.

See Also: BEGIN SEQUENCE ERRORLEVEL() RETURN

AEVAL()
Execute a code block for each element in an array

Syntax

AEVAL(<aArray>, <bBlock>,
[<nStart>], [<nCount>]) --> aArray

Arguments

<aArray> is the array to traverse.

<bBlock> is a code block to execute for each element encountered.

<nStart> is the starting element. If not specified, the default is element one.

<nCount> is the number of elements to process from <nStart>. If not specified, the default is all elements to the end of the array.

Returns

AEVAL() returns a reference to <aArray>.

Description

AEVAL() is an array function that evaluates a code block once for each element of an array, passing the element value and the element index as block parameters. The return value of the block is ignored. All elements in <aArray> are processed unless either the <nStart> or the <nCount> argument is specified.

AEVAL() makes no assumptions about the contents of the array elements it is passing to the block. It is assumed that the supplied block knows what type of data will be in each element.

AEVAL() is similar to DBEVAL() which applies a block to each record of a database file. Like DBEVAL(), AEVAL() can be used as a primitive for the construction of iteration commands for both simple and complex array structures.

Refer to the Code Blocks section in the Basic Concepts chapter of the Programming and Utilities guide for more information on the theory and syntax of code blocks.

Examples

. This example uses AEVAL() to display an array of filenames and file sizes returned from the DIRECTORY() function:

```
#include "Directry.ch"
//
LOCAL aFiles := DIRECTORY("*.dbf"), nTotal := 0
AEVAL(aFiles,;
   { | aDbfFile |;
      QOUT(PADR(aDbfFile[F_NAME], 10), aDbfFile[F_SIZE]),;
      nTotal += aDbfFile[F_SIZE]);
   } )
//
?
? "Total Bytes:", nTotal
```

. This example uses AEVAL() to build a list consisting of selected items from a multidimensional array:

```
#include "Directry.ch"
//
```

```
LOCAL aFiles := DIRECTORY("*.dbf"), aNames := {}
AEVAL(aFiles,;
   { | file | AADD(aNames, file[F_NAME]) };
   )
```

. This example changes the contents of the array element depending on a condition. Notice the use of the codeblock parameters: */

```
LOCAL aArray[6]
AFILL(aArray,"old")
AEVAL(aArray,;
{|cValue,nIndex| IF(cValue == "old",;
                aArray[nIndex] := "new",)})
```

Files: Library is CLIPPER.LIB.

See Also: DBEVAL() EVAL() QOUT()

READ
Activate full-screen editing mode using Get objects

Syntax

READ [SAVE]

Arguments

SAVE retains the contents of the current GetList after the READ terminates. Later, you can edit the same Get objects by issuing another READ. If not specified, the current GetList is assigned an empty array deleting all of the previous Get objects when the READ terminates.

Description

READ executes a full-screen editing mode using all Get objects created and added to the current GetList since the most recent CLEAR, CLEAR GETS, CLEAR ALL or READ commands. If there is a format procedure active, READ executes that procedure before entering the full-screen editing mode.

Within a READ, the user can edit the buffer of each Get object as well as move from one Get object to another. Before the user can enter a Get object, control passes to the associated WHEN <lPreCondition> if one has been assigned to that Get object. If <lPreCondition> returns true (.T.), the user is allowed to edit the buffer of the Get object. Otherwise control passes to the next Get object in the GetList. Within a GET buffer, the user can edit using the full complement of editing and navigation keys. See the tables below.

When the user presses a GET exit key, control passes to the associated RANGE or VALID postcondition if one has been specified. If either condition returns true (.T.) editing of the Get object is terminated and control passes to the next Get object. Otherwise, control remains within the current Get object until a valid value is entered or the user presses the Esc key.

When the user successfully enters a value into a Get object, the associated variable is assigned the value of the Get object's buffer.

The following tables list active keys within a READ:

READ Navigation Keys

Key	Action
Leftarrow, Ctrl-S	Character left. Does not move cursor to previous GET
Rightarrow, Ctrl-D	Character right. Does not move cursor to next GET
Ctrl-Leftarrow, Ctrl-A	Word left
Ctrl-Rightarrow, Ctrl-F	Word right
Uparrow, Shift-Tab, Ctrl-E	Previous GET
Dnarrow, Tab, Ctrl-X, Return, Ctrl-M	Next GET
Home	First character of GET
End	Last character of GET
Ctrl-Home	Beginning of first GET

READ Editing Keys

Key	Action
Del, Ctrl-G	Delete character at cursor position
Backspace, Ctrl-H	Destructive backspace
Ctrl-T	Delete word right

Ctrl-Y	Delete from cursor position to end of GET
Ctrl-U	Restore current GET to original value

READ Toggle Keys

Key	Action
Ins, Ctrl-V	Toggle insert mode

READ Exit Keys

Key	Action
Ctrl-W, Ctrl-C, PgUp, PgDn	Terminate READ saving current GET
Return, Ctrl-M	Terminate READ from last GET
Esc	Terminate READ without saving current GET
Uparrow	Terminate READ from first GET if READEXIT()=.T.
Dnarrow	Terminate READ from last GET if READEXIT()=.T.

Notes

- **Nested READs:** To perform a nested READ within a SET KEY, VALID, or WHEN procedure or user-defined function, declare or create a new GetList, perform a series of @...GET statements, then READ. When the procedure terminates, the new GetList is released and the previous GetList becomes visible again. See the example below.

- **Quick keys:** Pressing Home or End in quick succession goes to the first or last nonblank character in a Get object's buffer.

- **Terminating a READ:** A READ is terminated by executing a BREAK, CLEAR, CLEAR GETS, or CLEAR ALL from within a SET KEY procedure or a user-defined function initiated by VALID.

- **UPDATED():** If any Get object buffer was changed during the current READ, UPDATED() is set to true (.T.).

Examples

- This example defines several GETs then READs them:

```
CLEAR
cVar1 := cVar2 := cVar3 := SPACE(10)
@ 10, 10 SAY "Variable one:" GET cVar1 VALID ;
      !EMPTY(cVar1)
@ 11, 10 SAY "Variable two:" GET cVar2 ;
         WHEN RTRIM(cVar1) !=  "One"
@ 12, 10 SAY "Variable three:" GET cVar3 VALID ;
         !EMPTY(cVar3)
READ
```

- This example performs a nested READ within a SET KEY, WHEN, or VALID procedure or user-defined function:

```
LOCAL cName := SPACE(10)
@ 10, 10 GET cName VALID SubForm( cName )
READ
RETURN

FUNCTION SubForm( cLookup )
   LOCAL GetList := {}        // Create new GetList
   USE Sales INDEX Salesman NEW
   SEEK cLookup
```

```
IF FOUND()
   @ 15, 10 GET Salesman   // Add Get objects to
   @ 16, 10 GET Amount     // new GetList
   READ                    // READ from new GetList
ENDIF
CLOSE Sales
RETURN .T.                 // Release new GetList
```

Files: Library is CLIPPER.LIB.

See Also: @...GET CLEAR GETS LASTKEY() READEXIT() READMODAL()

RECALL
Restore records marked for deletion

Syntax

RECALL [<scope>] [WHILE <lCondition>]
[FOR <lCondition>]

Arguments

<scope> is the portion of the current database file to RECALL. The default scope is the current record, or NEXT 1. If a condition is specified, the default scope becomes ALL.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to RECALL within the given scope.

Description

RECALL is a database command that restores records marked for deletion in the current work area. This is the inverse of the DELETE command. If DELETED is ON, RECALL can restore the current record or a specific record, if you specify a RECORD scope. Note that once you PACK a database file, all marked records have been physically removed from the file and cannot be recovered.

In a network environment, RECALLing the current record requires an RLOCK(). RECALLing several records requires an FLOCK() or EXCLUSIVE USE of the current database file. Refer to the Network Programming chapter in the Programming and Utilities guide for more information.

Examples

. This examples show the results of RECALL:

```
USE Sales NEW
//
DELETE RECORD 4
? DELETED()               // Result: .T.
//
RECALL
? DELETED()               // Result: .F.
```

Files: Library is CLIPPER.LIB.

See Also: DELETE DELETED() FLOCK() PACK RLOCK() SET DELETED

REINDEX
Rebuild open indexes in the current work area

Syntax

```
REINDEX
[EVAL <lCondition>]
[EVERY <nRecords>]]
```

Arguments

EVAL <lCondition> specifies a condition that is evaluated either for each record processed or at the interval specified by the EVERY clause. This clause is identical to the EVAL clause of the INDEX command, but must be respecified in order for the reindexing operation to be monitored since the value of <lCondition> is transient.

EVERY <nRecords> specifies a numeric expression that modifies how often EVAL is evaluated. When using EVAL, the EVERY option offers a performance enhancement by evaluating the condition for every nth record instead of for each record reindexed. The EVERY keyword is ignored if no EVAL condition is specified.

Description

REINDEX is a database command that rebuilds all open indexes in the current work area. When the reindexing operation finishes, all rebuilt indexes remain open, order is reset to one, and the record pointer is positioned to the first record in the controlling index. If any of the indexes were created with SET UNIQUE ON, REINDEX adds only unique keys to the index. If any of the indexes were created using a FOR condition, only those key values from records matching the condition are added to the index.

In a network environment, REINDEX requires EXCLUSIVE USE of the current database file. Refer to the Network Programming chapter in the Programming and Utilities guide for more information.

Caution! REINDEX does not recreate the header of the index file when it recreates the index. Because of this, REINDEX does not help if there is corruption of the file header. To guarantee a valid index, always use INDEX ON in place of REINDEX to rebuild damaged indexes

Notes

Index key order, UNIQUE status, and the FOR condition are known to the (.ntx) file and are, therefore, respected and maintained by REINDEX.

Examples

. This example REINDEXes the index open in the current work area:

```
USE Sales INDEX Salesman, Territory NEW
REINDEX
```

. This example REINDEXes using a progress indicator:

```
USE Sales INDEX Salesman, Territory NEW
REINDEX EVAL NtxProgress() EVERY 10

FUNCTION NtxProgress
LOCAL cComplete := LTRIM(STR((RECNO()/LASTREC()) * 100))
@ 23, 00 SAY "Indexing..." + cComplete + "%"
RETURN .T.
```

Files: Library is CLIPPER.LIB.

See Also: INDEX PACK SET INDEX USE

RELEASE
Delete public and private memory variables

Syntax

RELEASE <idMemvar list>
RELEASE ALL [LIKE | EXCEPT <skeleton>]

Arguments

<idMemvar list> is a list of private or public variables or arrays to delete.

ALL [LIKE|EXCEPT <skeleton>] defines the set of visible private memory variables to assign, or to exclude from assignment of, a NIL value. <skeleton> is the wildcard mask to specify a group of memory variables to delete. The wildcard characters supported are * and ?.

Description

RELEASE is a memory variable command that performs one of two actions depending on how it is specified. If RELEASE is specified with <idMemvar list>, the specified public and private memory variables and/or arrays are deleted from memory. Previous hidden instances (public or private variables defined in higher-level procedures) become accessible upon termination of the procedure where the variable was originally created.

If RELEASE is specified with any form of the ALL clause, private memory variables created at the current procedure level are assigned a NIL and not deleted until the current procedure or user-defined function terminates. Public variables are unaffected by this form of the RELEASE command. To release public variables, you must RELEASE them explicitly or use CLEAR MEMORY.

Local or static variables are not affected by the RELEASE command. Local variables are released automatically when the procedure or user-defined function where the variables were declared terminates. Static variables cannot be released since they exist for the duration of the program.

Files: Library is CLIPPER.LIB.

See Also: CLEAR MEMORY LOCAL PRIVATE PUBLIC QUIT

RENAME
Change the name of a file

Syntax

RENAME <xcOldFile> TO <xcNewFile>

Arguments

<xcOldFile> is the name of the file to rename including an extension and optionally preceded by a drive and/or path designator. <xcOldFile> can be specified as a literal string or a character expression enclosed in parentheses.

TO <xcNewFile> specifies the new filename including extension and optionally prefaced by a drive and/or path designator. <xcNewFile> can be specified as a literal string or a character expression enclosed in parentheses.

Description

RENAME is a file command that changes the name of a specified file to a new name. If the source directory is different from the target directory, the file moves to the new directory. RENAME does not use SET DEFAULT and SET PATH to locate <xcOldFile>. Instead, the <xcOldFile> is renamed only if it is located in the current DOS directory or in the specified path.

In the instance that either <xcNewFile> exists or is currently open, RENAME does nothing. To trap this condition as an error, use the FILE() function before executing the command. See the example below.

Warning! Files must be CLOSEd before renaming. Attempting to rename an open file will produce unpredictable results. When a database file is RENAMEd, remember that any associated memo (.dbt) file must also be RENAMEd. Failure to do so may compromise the integrity of your program.

Examples

. This example renames a file, checking for the existence of the target file before beginning the RENAME operation:

```
xcOldFile := "OldFile.txt"
xcNewFile := "NewFile.txt"
IF !FILE(xcNewFile)
   RENAME (xcOldFile) TO (xcNewFile)
   ELSE
? "File already exists"
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: COPY FILE CURDIR() ERASE FILE() FERASE() FRENAME() RUN

REPLACE
Assign new values to field variables

Syntax

```
REPLACE <idField> WITH <exp>
   [, <idField2> WITH <exp2>...]
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
```

Arguments

<idField> is the name of the field variable to assign a new value. If <idField> is prefaced with an alias, the assignment takes place in the designated work area.

WITH <exp> defines the value to assign to <idField>.

<scope> is the portion of the current database file to REPLACE. The default is the current record, or NEXT 1. Specifying a condition changes the default to ALL records in the current work area.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to REPLACE within the given scope.

Description

REPLACE is a database command that assigns new values to the contents of one or more field variables in the current record in the specified work areas. The target field variables can be character, date, logical, memo, or numeric. REPLACE performs the same function as the assignment operator (:=) except that it assumes that an unaliased reference is to a field variable. This means that you can assign new values to field variables using assignment statements provided that the field variable references are prefaced with an alias, the FIELD alias, or declared using the FIELD declaration statement.

The default scope of REPLACE is the current record unless a scope or condition is specified. If a scope or condition is specified, the replace operation is performed on each record matching the scope and/or condition.

Warning! When you REPLACE a key field, the index is updated and the relative position of the record pointer within the index is changed. This means that REPLACEing a key field with a scope or a condition may yield an erroneous result. To update a key field, SET ORDER TO 0 before the REPLACE. This insures that the record pointer moves sequentially in natural order. All open indexes, however, are updated if the key field is REPLACEd.

In a network environment, REPLACEing the current record requires an RLOCK(). REPLACEing with a scope and/or condition requires an FLOCK() or EXCLUSIVE USE of the current database file. If a field is being REPLACEd in another work area by specifying its alias, that record must also be locked with an RLOCK(). Refer to the Network Programming chapter in the Programming and Utilities guide for more information.

Examples

. This example shows a simple use of REPLACE:

```
USE Customer NEW
APPEND BLANK
USE Invoices NEW
APPEND BLANK
```

```
//
REPLACE Charges WITH Customer->Markup * Cost,;
   Custid WITH Customer->Custid,;
   Customer->TranDate WITH DATE()
```

. Using assignment statements in place of the REPLACE command looks like this:

```
FIELD->Charges := Customer->Markup * FIELD->Cost
FIELD->Custid := Customer->Custid
Customer->TranDate := DATE()
```

Files: Library is CLIPPER.LIB.

See Also: COMMIT FLOCK() RLOCK()

REPORT FORM
Display a report to the console

Syntax

REPORT FORM <xcReport>
[TO PRINTER] [TO FILE <xcFile>] [NOCONSOLE]
[<scope>] [WHILE <lCondition>] [FOR <lCondition>]
[PLAIN | HEADING <cHeading>] [NOEJECT] [SUMMARY]

Arguments

<xcReport> is the name of the report form (.frm) file that contains the definition of the REPORT. If an extension is not specified, (.frm) is assumed. <xcReport> can be specified as a literal string or as a character expression enclosed in parentheses.

TO PRINTER echoes output to the printer.

TO FILE <xcFile> echoes output, without formfeed characters (ASCII 12), to a file. If a file extension is not specified, a (.txt) is added. You can specify <xcFile> as a literal string or as a character expression enclosed in parentheses.

NOCONSOLE suppresses all REPORT FORM output to the console. If not specified, output automatically displays to the console unless SET CONSOLE is OFF.

<scope> is the portion of the current database file to report. The default scope is ALL.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to report within the given scope.

PLAIN suppresses the display of the date and page number, and causes the report to print without page breaks. In addition, the report title and column headings display only at the top of the report.

HEADING places the result of <cHeading> on the first line of each page. <cHeading> is evaluated only once at the beginning of the report, before the record pointer is moved. If both PLAIN and HEADING are specified, PLAIN takes precedence.

NOEJECT suppresses the initial page eject when the TO PRINTER clause is used.

SUMMARY causes the REPORT FORM to display only group, subgroup, and grand total lines. Detail lines are suppressed.

Description

REPORT FORM is a console command that sequentially accesses records in the current work area displaying a tabular, and optionally grouped report with page and column headings from a definition held in a (.frm) file. The actual REPORT FORM file (.frm) is created using RL.EXE or dBASE III PLUS. Refer to the Report and Label Utility chapter in the Programming and Utilities guide for more information about creating report definitions.

When invoked, REPORT FORM sends output to the screen and, optionally, to the printer and/or a file. To suppress output to the screen while printing or echoing output to a file, SET CONSOLE OFF before the REPORT FORM invocation.

When invoked, REPORT FORM searches the current SET PATH drive and directory if the <xcReport> file is not found in the current directory and the path is not specified.

Notes

- **Interrupting REPORT FORM:** To allow the user to interrupt a REPORT FORM, use INKEY(), to test for an interrupt key press, as a part of the FOR condition . See the example below.

- **Printer margin:** REPORT FORM obeys the current SET MARGIN value for output echoed to the printer.

- **Forcing formfeed characters into an output file:** To include formfeed characters when sending a REPORT FORM TO FILE, redirect printer output to a file using SET PRINTER like this:

```
SET PRINTER TO <xcFile>
REPORT FORM <xcReport> TO PRINTER
SET PRINTER TO
```

- **Reporting in a network environment:** REPORT FORM commands executed in a network environment can be affected by changes made to database files by other users while the report is in progress. For example, if a user changes a key value from "A" to "Z" while the report is printing, the same record could appear twice.

Examples

- This example uses both a literal and an extended expression to execute a REPORT FORM:

```
LOCAL xcReport := "Sales"
USE Sales INDEX Sales NEW
REPORT FORM Sales TO PRINTER FOR Branch = "100";
          HEADING "Branch 100"
REPORT FORM (xcReport) TO PRINTER FOR Branch != "100"
```

- This example interrupts a REPORT FORM using INKEY() to test whether the user has pressed the Esc key:

```
#define K_ESC  27
USE Sales INDEX Sales NEW
REPORT FORM Sales WHILE INKEY() != K_ESC
```

Files: Library is CLIPPER.LIB.

See Also: LABEL FORM LIST

RESTORE
Retrieve memory variables from a memory (.mem) file

Syntax

RESTORE FROM <xcMemFile> [ADDITIVE]

Arguments

<xcMemFile> is the memory (.mem) file to load from disk. If an extension is not specified, the extension (.mem) is assumed. The filename may be specified as a literal string or as a character expression enclosed in parentheses.

ADDITIVE causes memory variables loaded from the memory file to be added to the existing pool of memory variables.

Description

RESTORE is a memory variable command that recreates public and private variables previously SAVEd to a memory (.mem) file and initializes them with their former values. The scope of the variable is not SAVEd with the variable but is instead established when the variable is RESTOREd. Arrays and local variables cannot be SAVEd or RESTOREd.

When memory variables are RESTOREd, they are recreated as private variables with the scope of the current procedure or user-defined function unless they exist as public variables and you specify the ADDITIVE clause . If ADDITIVE is specified, public and private variables with the same names are overwritten unless hidden with PRIVATE. If ADDITIVE is not specified, all public and private variables are released before the memory file is loaded.

Local and static variables are unaffected by RESTORE. If a local or static variable has been declared in the current procedure or user-defined function, and a variable with the same name is RESTOREd, only the local or static variable is visible unless references to the RESTOREd variable are prefaced with the MEMVAR alias.

Examples

. This example demonstrates a typical application of SAVE and RESTORE. Here memory variables containing screens are SAVEd TO and RESTOREd FROM memory files:

```
// Create and use a pseudoarray of screens
SAVE SCREEN TO cScreen1
SAVE ALL LIKE cScreen* TO Screens
//
<statements>...
//
RESTORE FROM Screens ADDITIVE
nNumber = "1"
RESTORE SCREEN FROM ("cScreen" + nNumber)
```

Files: Library is CLIPPER.LIB.

See Also: LOCAL PRIVATE PUBLIC SAVE

RESTORE SCREEN*
Display a saved screen

Syntax

RESTORE SCREEN [FROM <cScreen>]

Arguments

FROM <cScreen> specifies a character expression to display to the screen.

Description

RESTORE SCREEN is a command synonym for the RESTSCREEN() function that redisplays a previously saved screen, and is used with SAVE SCREEN to avoid repainting the original screen painted with @...SAY, @...GET, and ? commands.

RESTORE SCREEN works in two ways depending on whether or not you specify the FROM clause . If the you specify the FROM clause, the SCREEN is RESTOREd FROM <cScreen>. <cScreen> is a character expression, usually a variable assigned a screen image by SAVE SCREEN. If you do not specify the FROM clause, the SCREEN is RESTOREd from the default save screen buffer created by SAVE SCREEN specified without the TO clause.

SAVESCREEN() and RESTORESCREEN() functions supersede SAVE SCREEN and RESTORE SCREEN commands.

RESTORE SCREEN is a compatibility command and not recommended.

Warning! SAVE SCREEN, RESTORE SCREEN, SAVESCREEN(), and RESTSCREEN() are supported when using the default (IBM PC memory mapped) screen driver. Other screen drivers may not support saving and restoring screens.

Examples

. This example displays a small alert popup box using SAVE and RESTORE SCREEN:

```
IF FileAlert()
   COPY FILE Them.txt TO My.txt
ELSE
   BREAK
ENDIF
RETURN

FUNCTION FileAlert
   LOCAL lAnswer := .F., cScreen
   SAVE SCREEN TO cScreen
   @ 10, 10 CLEAR TO 12, 45
   @ 10, 10 TO 12, 45 DOUBLE
   @ 11, 12 SAY "File exists, overwrite? (y/n) ";
          GET lAnswer PICTURE "Y"
   READ
   RESTORE SCREEN FROM cScreen
   RETURN lAnswer
```

Files: Library is CLIPPER.LIB.

See Also: RESTORE RESTSCREEN() SAVE SAVESCREEN()

RUN
Execute a DOS command or program

Syntax

RUN | !* <xcCommandLine>

Arguments

<xcCommandLine> is any executable program including resident DOS commands and COMMAND.COM. It may be specified either as a literal string or as a character expression enclosed in parentheses.

Description

RUN executes a DOS command or program from within a compiled application. When you RUN a DOS program, CA-Clipper executes another copy of COMMAND.COM, passing the DOS command line at the same time. This has two implications. First, you must have enough memory for COMMAND.COM (27K for DOS 3.2) and the program you wish to execute. Second, COMMAND.COM must be available on the path specified by COMSPEC (the default is the root directory of the disk where you boot DOS). If COMMAND.COM is not located on this disk or the disk is changed, SET COMSPEC to the new location prior to running the CA-Clipper application. Note that SET DEFAULT and SET PATH have no effect on RUN.

The ! form of the RUN command is provided for compatibility purposes only and therefore not recommended.

Warning! Do not RUN memory-resident programs from within CA-Clipper since you may lose memory when the control returns to your application program.

Examples

. This example uses RUN with MEMOREAD() and MEMOWRIT() to create a user-defined function that calls a text editor with the current memo field:

```
lSuccess = EditorMemo("Qedit", "Notes")
RETURN

FUNCTION EditorMemo( cEditor, cMemofld )
   IF MEMOWRIT("Clipedit.tmp", &cMemofld.)
      RUN (cEditor + " Clipedit.tmp")
      REPLACE &cMemofld. WITH MEMOREAD("Clipedit.tmp")
      ERASE Clipedit.tmp
      RETURN .T.
   ELSE
      RETURN .F.
   ENDIF
```

. One of the options you may want to give your users is direct access to DOS. Do this with:

```
RUN COMMAND
```

To make it easier for the user to return to the application program, change the DOS prompt in the application batch file like this:

```
REM Application Batch File
ECHO OFF
PROMPT DOS Access: Type EXIT to return to ;
         application$_$p$g
<your application program>
PROMPT $p$g
```

Then, instruct the user to execute the application batch file in place of the application .EXE file.

Files: Library is CLIPPER.LIB.

SAVE
Save variables to a memory (.mem) file

Syntax

SAVE TO <xcMemFile> [ALL [LIKE | EXCEPT <skeleton>]]

Arguments

<xcMemFile> is the memory (.mem) file to SAVE to disk. You may specify the filename as a literal string or as a character expression enclosed in parentheses. If you specify no extension, the file is created with a (.mem) extension.

ALL [LIKE|EXCEPT <skeleton>] defines the set of visible private and public memory variables to save to <xcMemFile>. <skeleton> is the wildcard mask that characterizes a group of memory variables to SAVE. The wildcard characters supported are * and ?.

Description

SAVE copies public and private memory variables visible within the current procedure or user-defined function to a memory (.mem) file. Arrays and local and static variables, however, cannot be SAVEd. When variables are SAVEd, they are copied without any reference to scope. Variables hidden by PRIVATE or LOCAL declarations are not SAVEd.

If you specify the ALL LIKE clause, variable names matching the <skeleton> mask are saved. By contrast, if you specify ALL EXCEPT, variable names not matching the <skeleton> are saved.

You can specify a <skeleton> that includes wildcard characters. The wildcard character * matches any group of adjacent characters ending a variable name and can be specified only at the end of the <skeleton>. The wildcard character ? matches any single character and can be specified anywhere within the <skeleton>.

Examples

. This example saves all visible private and public variables to Temp.mem:

```
PRIVATE cOne := "1"
SAVE ALL TO Temp
```

. This example saves all visible private and public variables with names beginning with c to Myvars.mem:

```
SAVE ALL LIKE c* TO MyVars
```

. This example saves all visible private and public variables with names that do not begin with c to Myvars2.mem:

```
SAVE ALL EXCEPT c* TO MyVars2
```

Files: Library is CLIPPER.LIB.

See Also: LOCAL PRIVATE PUBLIC RESTORE

SAVE SCREEN*
Save current screen to a buffer or variable

Syntax

SAVE SCREEN [TO <idVar>]

Arguments

TO <idVar> specifies a variable to contain the current screen contents as a character value. If <idVar> is not visible or does not exist, a private memory variable is created and assigned to the screen.

Description

SAVE SCREEN is a command synonym for the SAVESCREEN() function that saves the screen from 0, 0 to MAXROW(), MAXCOL() in a the default screen buffer, or an optional variable. If the screen is saved to a variable, the variable can be any storage class including field, local, static, or an array element. Note, however, you cannot SAVE an array or local or static variable to (.mem) files to save multiple screens to disk.

SAVE SCREEN is used with RESTORE SCREEN to eliminate repainting an original screen that has been temporarily replaced. You may save multiple screens by assigning each screen to a separate variable.

SAVE SCREEN is a compatibility command and not recommended. It is superseded by the SAVESCREEN() function which can save partial or full screens.

Warning! SAVE SCREEN, RESTORE SCREEN, SAVESCREEN(), and RESTSCREEN() are supported when using the default (IBM PC memory mapped) screen driver. Other screen drivers may not support saving and restoring screens.

Examples

. This code skeleton uses a static array to store saved screens:

```
STATIC aScreens[10]
SAVE SCREEN TO aScreens[1]
//
<statements>...
//
RESTORE SCREEN FROM aScreens[1]
```

. This example saves and restores screens using a database file:

```
USE Screens INDEX Name NEW
APPEND BLANK

Screens->Name := "Screen001"          // Save the screen name
SAVE SCREEN TO Screens->Image         // Save a new screen image
//
<statements>...
//
SEEK "Screen001"                      // Find the screen
RESTORE SCREEN FROM Screens->Image    // Restore it
```

Files: Library is CLIPPER.LIB.

See Also: RESTORE RESTSCREEN() SAVE SAVESCREEN()

SEEK
Search an index for a specified key value

Syntax

SEEK <expSearch>

Arguments

<expSearch> is an expression to match with an index key.

Description

SEEK is a database command that searches the controlling index starting with the first key and proceeds until a match is found or there is a key value greater than <expSearch>. If there is a match, the record pointer is positioned to the record number found in the index. If SOFTSEEK is OFF (the default) and SEEK does not find a match, the record pointer is positioned to LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.). If SOFTSEEK is ON, the record pointer is positioned to the record with the first key value greater than the search argument and FOUND() returns false (.F.). In this case, EOF() only returns true (.T.) if there are no keys in the index greater than the search argument.

SET EXACT has no effect on the operation of SEEK.

Examples

. This example illustrates the SEEK command:

```
USE Sales INDEX Branch NEW
SEEK "100"
? FOUND(), EOF(), RECNO()      // Result: .T. .F. 1
```

. This user-defined function performs an exact SEEK for an index with a character key:

```
USE Invoice INDEX Invoice NEW
IF SeekExact("10001")
   ? "Invoice on file"
ELSE

   ? "Invoice not on file"
ENDIF
RETURN

FUNCTION SeekExact( cSearch )
   SEEK PADR(cSearch, LEN(&(INDEXKEY(0))))
   RETURN (FOUND())
```

Files: Library is CLIPPER.LIB.

See Also: EOF() FOUND() SET SOFTSEEK DBSEEK()

SELECT
Change the current work area

Syntax

SELECT <xnWorkArea> | <idAlias>

Arguments

<xnWorkArea> is the work area number between zero and 250 inclusive. This argument is an extended expression and can be specified either as a literal number or as a numeric expression enclosed in parentheses.

<idAlias> is the name of an existing work area to SELECT if there is a database file open in that area.

Description

SELECT is a database command that changes work areas. CA-Clipper supports 250 work areas, with each work area a logical handle to an open database file and all of its attributes. You can refer to work areas with SELECT by number or by alias. The alias of a work area is automatically assigned when a database file is USEd in that work area or by using the ALIAS clause.

Work area zero refers to the first empty or next available work area. Using this, you can SELECT 0 and USE <xcDatabase> as a method of opening database files.

Notes

. **Aliased expressions:** Aliased expressions are a much more powerful method of selecting new work areas than the SELECT command. Instead of SELECTing a work area then performing an operation for that work area, you can apply an alias to an expression that performs the operation. This is done by specifying the alias of the remote work area and the expression enclosed in parentheses. For example, to access the value of EOF() in an unselected work area, you would normally execute a series of statements like the following:

```
SELECT Remote
   ? EOF()

   SELECT Main
```

Using the aliased expression form, these statements become:

```
? Remote->(EOF())
```

. **USE...NEW:** Instead of using SELECT0 and USE <xcDatabase> to open a database file in a new work area, the preferred method is to USE <xcDatabase> NEW.

Examples

. This example opens a series of database files by SELECTing each work area by number then USEing each database file in that work area:

```
SELECT 1
USE Customer
SELECT 2
USE Invoices
SELECT 3
USE Parts
SELECT Customer
```

- A better method is to open each database in the next available work area by specifying the NEW clause on the USE command line. In this example USE...NEW is employed instead of SELECT0 then USE:

```
USE Customer NEW
USE Invoices NEW
SELECT Customer
```

- This code fragment changes work areas while saving the current work area name to a variable using the SELECT() function. After executing an operation for the new work area, the original work area is restored:

```
nLastArea := SELECT()
USE Newfile NEW
//
<statements>...
//
SELECT (nLastArea)
```

Files: Library is CLIPPER.LIB.

See Also: ALIAS() SELECT() SET INDEX USE USED()

SET ALTERNATE
Echo console output to a text file

Syntax

SET ALTERNATE TO [<xcFile> [ADDITIVE]]
SET ALTERNATE on | OFF | <xlToggle>

Arguments

TO <xcFile> opens a standard ASCII text file for output with a default extension of (.txt). The filename may optionally include an extension, drive letter, and/or path. You may specify <xcFile> either as a literal filename or as a character expression enclosed in parentheses. Note that if a file with the same name exists, it is overwritten.

ADDITIVE causes the specified alternate file to appended to instead of overwritten. If not specified, the specified alternate file is truncated before new information is written to it.

ON causes console output to be written to the open text file.

OFF discontinues writing console output to the text file without closing the file.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET ALTERNATE is a console command that lets you write the output of console commands to a text file. Commands such as LIST, REPORT FORM, LABEL FORM, and ? that display to the screen without reference to row and column position are console commands. Most of these commands have a TO FILE clause that performs the same function as SET ALTERNATE. Full-screen commands such as @...SAY cannot be echoed to a disk file using SET ALTERNATE. Instead you can use SET PRINTER TO <xcFile> with SET DEVICE TO PRINTER to accomplish this.

SET ALTERNATE has two basic forms. The TO <xcFile> form creates a DOS text file with a default extension of (.txt) and overwrites any other file with the same name. Alternate files are not related to work areas with only one file open at a time. To close an alternate file, use CLOSE ALTERNATE, CLOSE ALL, or SET ALTERNATE TO with no argument.

The on|OFF form controls the writing of console output to the current alternate file. SET ALTERNATE ON begins the echoing of output to the alternate file. SET ALTERNATE OFF suppresses output to the alternate file but does not close it.

Examples

. This example creates an alternate file and writes the results of the ? command to the file for each record in the Customer database file:

```
SET ALTERNATE TO Listfile
SET ALTERNATE ON

USE Customer NEW
DO WHILE !EOF()
   ? Customer->Lastname, Customer->City
   SKIP
ENDDO
SET ALTERNATE OFF
```

```
CLOSE ALTERNATE
CLOSE Customer
```

Files: Library is CLIPPER.LIB.

See Also: CLOSE FOPEN() FWRITE() SET CONSOLE SET PRINTER

SET BELL
Toggle automatic sounding of the bell during full-screen operations

---

Syntax

SET BELL on | OFF | <xlToggle>

Arguments

ON enables the BELL.

OFF disables the BELL.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET BELL is an environment command that toggles the sound of the bell. If SET BELL is ON, the bell sounds in the following situations:

. The user enters a character at the last position in a GET.

. The user attempts to enter invalid data into a GET. The data is validated by the data type of the GET variable, the PICTURE template, and by the RANGE clause. Violating a VALID condition does not sound the bell, regardless of the SET BELL status.

To sound the bell explicitly, you can use either ?? CHR(7) or the TONE() function. TONE() is perhaps more useful since you can vary both the pitch and duration of the sound.

Files: Library is CLIPPER.LIB.

See Also: @...GET CHR() SET CONFIRM TONE()

AFIELDS()*
Fill arrays with the structure of the current database file

Syntax

AFIELDS([<aFieldNames>], [<aTypes>],
[<aWidths>], [<aDecimals>]) --> nFields

Arguments

<aFieldNames> is the array to fill with field names. Each element is a character string.

<aTypes> is the array to fill with the type of fields in <aFieldNames>. Each element is a character string.

<aWidths> is the array to fill with the widths of fields in <aFieldNames>. Each element is numeric data type.

<aDecimals> is the array to fill with the number of decimals defined for fields in <aFieldNames>. Each element is numeric data type. If the field type is not numeric, the <aDecimals> element is zero.

Returns

AFIELDS() returns the number of fields or the length of the shortest array argument, whichever is less. If no arguments are specified, or if there is no file in USE in the current work area, AFIELDS() returns zero.

Description

AFIELDS() is an array function that fills a series of arrays (structure attribute arrays) with the structure of a database file currently open, one element in each array per field. AFIELDS() works like ADIR(), filling a series of existing arrays with information. To use AFIELDS(), you must first create the arrays to hold the database structure information, each with the same number of elements as the number of fields (i.e. FCOUNT()). Once the structure attribute arrays are created, you can then invoke AFIELDS() to fill the structure arrays with information about each field.

By default, AFIELDS() operates on the currently selected work area. It can operate on an unselected work area if you specify it within an aliased expression (see example below).

AFIELDS() is a compatibility function and therefore is not recommended. It is superseded by DBSTRUCT() which does not require the existence of any arrays prior to invocation and returns a multidimensional array containing the current database file structure.

Examples

. This example demonstrates how AFIELDS() and ACHOICE() can be used together to create a fields picklist:

```
USE Sales NEW
PRIVATE aFieldNames[FCOUNT()]
AFIELDS(aFieldNames)
@ 1, 0 TO 10, 10 DOUBLE
nChoice := ACHOICE(2, 1, 9, 9, aFieldNames)
@ 12, 0 SAY IF(nChoice != 0, aFieldNames[nChoice],;
               "None selected")
RETURN
```

. This example uses AFIELDS() with an aliased expression to fill arrays with the structure of Sales.dbf, open in an unselected work

```
area:

LOCAL aFieldNames, aTypes, aWidths, aDecimals
USE Sales NEW
USE Customer NEW
//
aFieldNames := Sales->(ARRAY(FCOUNT()))
aTypes := Sales->(ARRAY(FCOUNT()))
aWidths := Sales->(ARRAY(FCOUNT()))
aDecimals := Sales->(ARRAY(FCOUNT()))
//
Sales->(AFIELDS(aFieldNames, aTypes, ;
        aWidths, aDecimals))
```

Files: Library is EXTEND.LIB.

See Also: ACHOICE() ADIR()* AEVAL() ASCAN() DBCREATE() DBSTRUCT()

SET CENTURY
Modify the date format to include or omit century digits

Syntax

SET CENTURY on | OFF | <xlToggle>

Arguments

ON allows input and display of the century digits for dates.

OFF suppresses the input and display of the century digits for dates.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET CENTURY modifies the current date format as set by SET DATE. SET CENTURY ON changes the date format to contain four digits for the year. With the date format set to four digits for the year, date values display with a four-digit year, and dates of any century can be input.

SET CENTURY OFF changes the date format to contain only two digits for the year. With the date format set to only two digits for the year (CENTURY OFF), the century digits of dates are not displayed and cannot be input.

Note that only the display and input format of dates is affected; date calculations maintain the century information regardless of the date format.

CA-Clipper supports all dates in the range 01/01/0100 to 12/31/2999.

Examples

. This example shows the results of a simple SET CENTURY command:

```
SET CENTURY OFF
? DATE()                  // Result: 09/15/90
SET CENTURY ON
? DATE()                  // Result: 09/15/1990
```

Files: Library is CLIPPER.LIB.

See Also: CTOD() DATE() DTOC() SET DATE SET EPOCH

SET COLOR*
Define screen colors

Syntax

SET COLOR | COLOUR TO [[<standard>]
[,<enhanced>] [,<border>] [,<background>]
[,<unselected>]] | (<cColorString>)

Arguments

<standard> is the color that paints all console, full-screen, and interface commands and functions when displaying to the screen. This includes commands such as @...PROMPT, @...SAY, and ?; and functions such as ACHOICE(), DBEDIT(), and MEMOEDIT().

<enhanced> is the color that paints highlighted displays. This includes GETs with INTENSITY ON, the MENU TO, DBEDIT(), and ACHOICE() selection highlight.

<border> is the color that paints the area around the screen that cannot be written to.

<background> is not currently supported by any machines for which Computer Associates provides drivers. This setting is supplied for compatibility purposes only.

<unselected> is a color pair that provides input focus by displaying the current GET in the enhanced color while other GETs are displayed in this color.

<cColorString> is a character string enclosed in parentheses containing the color settings. This facility lets you specify the color settings as an expression in place of a literal string or macro variable.

SET COLOR TO with no argument restores the default colors to W/N, N/W, N, N, N/W.

Description

SET COLOR, a command synonym for the SETCOLOR() function, defines colors for subsequent screen painting activity. Each SET COLOR command specifies a list of color settings for the five types of screen painting activity. Each setting is a foreground and background color pair separated by the slash (/) character. Foreground defines the color of characters displayed on the screen. Background defines the color displayed behind the character. Spaces and nondisplay characters display as background only.

In addition to color, a foreground setting can have an attribute, high intensity or blinking. With a monochrome display, high intensity enhances brightness of painted text. With a color display, high intensity changes the hue of the specified color making it a different color. For example, N displays foreground text as black where N+ displays the same text as gray. High intensity is denoted by +. The blinking attribute causes the foreground text to flash on and off at a rapid interval. Blinking is denoted with *. An attribute character can occur anywhere in a setting, but is always applied to the foreground color regardless of where it occurs.

Each color can be specified using either a letter or a number, but numbers and letters cannot be mixed within a setting. Note that numbers are supplied for compatibility purposes and are not recommended.

All settings are optional. If a setting is skipped, its previous value is retained with only new values set. Skipping a foreground or

background color within a setting sets the color to black.

The following colors are supported:

Color Table

Color	Letter	Number	Monochrome
Black	N, Space	0	Black
Blue	B	1	Underline
Green	G	2	White
Cyan	BG	3	White
Red	R	4	White
Magenta	RB	5	White
Brown	GR	6	White
White	W	7	White
Gray	N+	8	Black
Bright Blue	B+	9	Bright Underline
Bright Green	G+	10	Bright White
Bright Cyan	BG+	11	Bright White
Bright Red	R+	12	Bright White
Bright Magenta	RB+	13	Bright White
Yellow	GR+	14	Bright White
Bright White	W+	15	Bright White
Black	U		Underline
Inverse Video	I		Inverse Video
Blank	X		Blank

SET COLOR is a compatibility command and not recommended. It is superseded by the SETCOLOR() function which can return the current color as well as set a new color.

Notes

- **Monochrome monitors:** On monochrome monitors, color is not supported. CA-Clipper, however, supports the monochrome attributes inverse video (I) and underlining (U).

- **Screen drivers:** SET COLOR TO, using numbers, may not be supported by other than the default screen driver.

Examples

- This example uses the unselected setting to make the current GET red on white while the rest are black on white:

```
cColor:= "W/N,R/W,,,N/W"
SET COLOR TO (cColor)
cOne := cTwo := SPACE(10)

@ 1, 1 SAY "Enter One: " GET cOne
@ 2, 1 SAY "Enter Two: " GET cTwo
READ
```

- In this example a user-defined function gets a password from the user using the blank (X) enhanced setting to hide the password as the user types:

```
IF !DialogPassWord(12, 13, "W+/N", "FUNSUN", 3)
   ? "Sorry, your password failed"
   QUIT
ENDIF

FUNCTION DialogPassWord( nRow, nCol, ;
       cStandard, cPassword, nTries )
   LOCAL nCount := 1, cColor := SETCOLOR()
   SET COLOR TO (cStandard + ", X")      // Blank input
```

```
//
DO WHILE nCount < nTries
   cUserEntry:= SPACE(6)
   @ nRow, nCol SAY  "Enter password: " GET ;
            cUserEntry
   READ
   //
   IF LASTKEY() == 27
      SET COLOR TO (cColor)
      RETURN .F.
   ELSEIF cUserEntry == cPassword
      SET COLOR TO (cColor)
      RETURN .T.
   ELSE
      nCount++
   ENDIF
ENDDO
//
SET COLOR TO (cColor)
RETURN .F.
```

Files: Library is CLIPPER.LIB.

See Also: @...GET @...SAY ISCOLOR() SETCOLOR() SETBLINK() Colors

SET CONFIRM
Toggle required exit key to terminate GETs

Syntax

SET CONFIRM on | OFF | <xlToggle>

Arguments

ON requires the user to press an exit key to leave a GET.

OFF allows the user to leave a GET by typing past the end without pressing an exit key.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET CONFIRM determines whether an exit key is required to leave a GET. If CONFIRM is OFF, the user can type past the end of a GET and the cursor will move to the next GET, if there is one. If there isn't another GET, the READ terminates. If, however, CONFIRM is ON, an exit key must be pressed to leave the current GET.

In all cases, attempting to leave the current GET executes the RANGE or VALID clauses, unless the user presses the Esc key.

See @...GET for more information on the behavior of GETs.

Files: Library is CLIPPER.LIB.

See Also: @...GET READ SET BELL

SET CONSOLE
Toggle console display to the screen

Syntax

SET CONSOLE ON | off | <xlToggle>

Arguments

ON displays the output of console commands on the screen.

OFF suppresses the screen display of console commands.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET CONSOLE determines whether or not console commands send output to the screen. Console commands are commands that display to the screen without reference to row and column position. In addition to sending output to the screen, console commands can simultaneously send output to the printer and/or a DOS text file. Output is sent to the printer using the TO PRINTER clause common to many console commands, or with the SET PRINTER ON command. Output is sent to a file using the TO FILE clause, SET ALTERNATE, or SET PRINTER TO.

With CONSOLE ON, console commands display to the screen. With CONSOLE OFF, the screen display of console commands is suppressed, but the echoing of output to either a file or the printer is unaffected. This lets you send the output of console commands such as REPORT and LABEL FORM to the printer without the screen display--a common occurrence.

Notes

- **Keyboard input:** For console commands that accept input (including ACCEPT, INPUT, and WAIT), SET CONSOLE affects the display of the prompts as well as the input areas. As a consequence, a SET CONSOLE OFF before one of these commands will not only prevent you from seeing what you type, but will also prevent the display of the message prompt.

- **Full-screen commands:** Full-screen commands such as @...SAY display to the screen independent of the current CONSOLE SETting. For this category of output commands, device control is performed using SET DEVICE to control whether output goes to the screen or printer, and SET PRINTER TO to echo output to a file.

Examples

- This example uses REPORT FORM to output records to the printer while suppressing output to the screen:

```
USE Sales NEW
SET CONSOLE OFF
REPORT FORM Sales TO PRINTER
SET CONSOLE ON
```

Files: Library is CLIPPER.LIB.

See Also: SET DEVICE

SET CURSOR
Toggle the screen cursor on or off

Syntax

SET CURSOR ON | off | <xlToggle>

Arguments

ON enables the cursor display.

OFF disables the cursor display.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET CURSOR toggles the screen cursor on or off. When the CURSOR is OFF, keyboard entry and screen display are unaffected. The cursor is merely hidden and data entry may still be accomplished without the cursor being visible. ROW() and COL() are updated as if the cursor was visible.

This command suppresses the cursor while the screen is being painted. Ideally, the only time the cursor shows in a production program is when the user is editing GETs, MEMOEDIT(), or some kind of line edit.

Examples

. This example shows a typical use of SET CURSOR:

```
LOCAL lAnswer := .F.
@ 24, 0
@ 24, 15 SAY "Do you want to QUIT? [Y/N]";
   GET lAnswer;
   PICTURE "Y"
SET CURSOR ON
READ
SET CURSOR OFF
```

Files: Library is CLIPPER.LIB.

See Also: SET CONSOLE SETCURSOR() SETPOS()

SET DATE
Set the date format for input and display

Syntax

SET DATE FORMAT [TO] <cDateFormat>
SET DATE [TO] AMERICAN | ansi | british | french
| german | italian | japan | usa

Arguments

<cDateFormat> is a character expression that directly specifies the date format when the FORMAT clause is specified. <cDateFormat> must evaluate to a string of 12 or fewer characters.

When specified, <cDateFormat> is analyzed to determine the proper placement and number of digits for the day, month, and year digits. The position of the day, month, and year digits is determined by scanning the string for one or more occurrences of the letters d, m, and y, respectively. Other characters in the string are copied verbatim into displayed date values.

When FORMAT is not used, one of several keywords describes the date format. The following table shows the format for each keyword setting:

SET DATE Formats

SETting	Format
AMERICAN	mm/dd/yy
ANSI	yy.mm.dd
BRITISH	dd/mm/yy
FRENCH	dd/mm/yy
GERMAN	dd.mm.yy
ITALIAN	dd-mm-yy
JAPAN	yy/mm/dd
USA	mm-dd-yy

Description

SET DATE is an environment command that sets the display format for date values. SET DATE is a global setting that affects the behavior of dates throughout a program, allowing you to control date formatting in a way that facilitates porting applications to foreign countries.

Examples

. In this example the FORMAT clause directly specifies the date format:

```
SET DATE FORMAT "yyyy:mm:dd"
```

. This example configures the date setting at runtime by passing a DOS environment variable to the program, retrieving its value with GETENV(), and setting DATE with the retrieved value:

```
C>SET CLIP_DATE=dd/mm/yy
```

In the configuration section of the application program, the date format is set like this:

```
FUNCTION AppConfig
   SET DATE FORMAT TO GETENV("CLIP_DATE")
   RETURN NIL
```

Files: Library is CLIPPER.LIB.

See Also: CTOD() DATE() DTOC() DTOS() SET CENTURY SET EPOCH

SET DECIMALS
Set the number of decimal places displayed

Syntax

SET DECIMALS TO [<nDecimals>]

Arguments

TO <nDecimals> is the number of decimal places to display. The default value is two.

SET DECIMALS TO with no argument is equivalent to SET DECIMALS TO 0.

Description

SET DECIMALS determines the number of decimal places displayed in the results of numeric functions and calculations. Its operation depends directly on the FIXED setting. If FIXED is OFF, SET DECIMALS establishes the minimum number of decimal digits displayed by EXP(), LOG(), SQRT(), and division operations. If FIXED is ON, all numeric values are displayed with exactly the number of decimal places specified by SET DECIMALS. Note that neither SET DECIMALS nor SET FIXED affects the actual numeric precision of calculations--only the display format is affected.

To provide finer control of numeric display, you can use the PICTURE clause of @...SAY, @...GET, and the TRANSFORM() function.

Examples

. These examples show various results of the SET DECIMALS command:

```
SET FIXED ON
SET DECIMALS TO 2            // The default setting
? 2/4                        // Result: 0.50
? 1/3                        // Result: 0.33
SET DECIMALS TO 4
? 2/4                        // Result: 0.5000
? 1/3                        // Result: 0.3333
```

Files: Library is CLIPPER.LIB.

See Also: @...GET @...SAY SET FIXED TRANSFORM()

SET DEFAULT
Set the CA-Clipper default drive and directory

Syntax

SET DEFAULT TO [<xcPathspec>]

Arguments

TO <xcPathspec> identifies a disk drive and the directory as the default and can be specified either as a literal path specification or as a character expression enclosed in parentheses. If you specify both a drive and directory, a colon must be included after the drive letter.

SET DEFAULT TO specified without an argument defaults to the current DOS drive and directory.

Description

SET DEFAULT sets the drive and directory where the application program creates and saves files, with the exception of temporary files and files created with the low-level file functions.

SET DEFAULT does not change the DOS drive and directory. When attempting to access files, the DEFAULT drive and directory are searched first. To set additional search paths for file access, use SET PATH.

Notes

- **Initial Default:** When a CA-Clipper program starts, the default drive and directory are the current DOS drive and directory. Within the program, you can change this with SET DEFAULT.

- **Running external programs:** Executing a RUN command accesses the current DOS drive and directory.

Examples

- This example shows a typical use of SET DEFAULT:

```
SET PATH TO
? FILE("Sales.dbf")      // Result: .F.
//
SET DEFAULT TO C:\CLIPPER\FILES
? FILE("Sales.dbf")      // Result: .T.
//
SET DEFAULT TO C:        // Change default drive
SET DEFAULT TO \         // Change to root directory
SET DEFAULT TO ..        // Change to parent directory
```

Files: Library is CLIPPER.LIB.

See Also: CURDIR() SET PATH

SET DELETED
Toggle filtering of deleted records

Syntax

SET DELETED on | OFF | <xlToggle>

Arguments

ON ignores deleted records.

OFF processes deleted records.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET DELETED toggles automatic filtering of records marked for deletion in all work areas. When SET DELETED is ON, most commands ignore deleted records. If, however, you refer to a record by record number (GOTO or any command that supports the RECORD scope) the record is not ignored even if marked for deletion. Additionally, SET DELETED ON has no effect on INDEX or REINDEXing.

RECALL ALL honors SET DELETED and does not recall any records.

Notes

. **Filtering deleted records in a single work area:** To confine the filtering of deleted records to a particular work area, SELECT the work area, then SET FILTER TO !DELETED().

Examples

. This example illustrates the effect of using SET DELETED:

```
USE Sales NEW
? LASTREC()                  // Result: 84
//
DELETE RECORD 4
COUNT TO nCount
? nCount                     // Result: 84
//
SET DELETED ON
COUNT TO nCount
? nCount                     // Result: 83
```

Files: Library is CLIPPER.LIB.

See Also: DELETE DELETED() RECALL SET FILTER

SET DELIMITERS
Toggle or define GET delimiters

Syntax

SET DELIMITERS on | OFF | <xlToggle>
SET DELIMITERS TO [<cDelimiters> | DEFAULT]

Arguments

ON displays delimiters for GET variables.

OFF suppresses the delimiter display.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

TO <cDelimiters> defines a one or two character delimiter. Specifying a single character uses the same character as both the beginning and ending delimiter. Specifying two characters uses the first as the beginning delimiter and the second as the ending delimiter.

TO DEFAULT or no delimiters SETs the delimiters to colons which are the default delimiters.

Description

SET DELIMITERS is a dual purpose command that both defines characters used to delimit GETs and toggles the automatic display of delimiters ON or OFF. The @...GET command can display delimiters that surround a GET object's display. If DELIMITERS is ON, the delimiters add two characters to the length of the Get object display.

You can configure the delimiter characters using the TO <cDelimiters> clause. The DEFAULT delimiter character is the colon (:). When specifying delimiters, the beginning and ending delimiter characters can be different. If you wish to suppress either the right, left, or both delimiters, use a space instead of the delimiter character.

Typically, delimiters are unnecessary since GETs display in reverse video or enhanced color if INTENSITY is ON.

Examples

. This example SETs DELIMITERS TO a colon and a space for the first GET and the square bracket characters for the second:

```
LOCAL cVar := SPACE(5), cVar2 := SPACE(5)
SET DELIMITERS ON
SET DELIMITERS TO ": "
@ 1, 0 SAY "Enter" GET cVar
SET DELIMITERS TO "[]"
@ 2, 0 SAY "Enter" GET cVar2
READ
```

Files: Library is CLIPPER.LIB.

See Also: @...GET SET INTENSITY

SET DEVICE
Direct @...SAYs to the screen or printer

Syntax

SET DEVICE TO SCREEN | printer

Arguments

TO SCREEN directs all @...SAYs to the screen and is independent of the SET PRINTER and CONSOLE settings.

TO PRINTER directs all @...SAYs to the device set with SET PRINTER TO. This can include a local printer port, a network spooler, or a file.

Description

SET DEVICE directs the output of @...SAY commands to either the screen or the printer. When DEVICE is SET TO PRINTER, @...SAY commands are sent to the printer and not echoed to the screen. In addition, @...SAY commands observe the current SET MARGIN value.

When sending @...SAYs to the printer, CA-Clipper performs an automatic EJECT whenever the current printhead row position is less than the last print row position. An EJECT resets PCOL() and PROW() values to zero. To reset PCOL() and PROW() to new values, use the SETPRC() function.

To send @...SAYs to a file, use SET PRINTER TO <xcFile> with SET DEVICE TO PRINTER.

Examples

. This example directs @...SAYs to the printer:

```
SET DEVICE TO PRINTER
@ 2,10 SAY "Hello there"
EJECT
```

. This example directs @...SAYs to a file:

```
SET PRINTER TO Output.txt
SET DEVICE TO PRINTER
@ 10, 10 SAY "File is: Output.txt"
@ 11, 10 SAY DATE()
SET PRINTER TO            // Close the output file
SET DEVICE TO SCREEN
```

Files: Library is CLIPPER.LIB.

See Also: @...SAY EJECT PCOL() PROW() SET PRINTER SETPRC()

SET EPOCH
Control the interpretation of dates with no century digits

Syntax

SET EPOCH TO <nYear>

Arguments

TO <nYear> specifies the base year of a 100-year period in which all dates containing only two year digits are assumed to fall.

Description

SET EPOCH is an environment command that determines the interpretation of date strings containing only two year digits. When such a string is converted to a date value, its year digits are compared with the year digits of <nYear>. If the year digits in the date are greater than or equal to the year digits of <nYear>, the date is assumed to fall within the same century as <nYear>. Otherwise, the date is assumed to fall in the following century.

The default value for SET EPOCH is 1900, causing dates with no century digits to be interpreted as falling within the twentieth century.

CA-Clipper supports all dates in the range 01/01/0100 to 12/31/2999.

Examples

. This example shows the effects of SET EPOCH:

```
SET DATE FORMAT TO "mm/dd/yyyy"
? CTOD("05/27/1904")          // Result: 05/27/1904
? CTOD("05/27/67")            // Result: 05/27/1967
? CTOD("05/27/04")            // Result: 05/27/1904
//
SET EPOCH TO 1960
? CTOD("05/27/1904")          // Result: 05/27/1904
? CTOD("05/27/67")            // Result: 05/27/1967
? CTOD("05/27/04")            // Result: 05/27/2004
```

Files: Library is CLIPPER.LIB.

See Also: CTOD() DATE() DTOC() DTOS() SET CENTURY SET DATE

SET ESCAPE
Toggle Esc as a READ exit key

Syntax

SET ESCAPE ON | off | <xlToggle>

Arguments

ON enables Esc as a as a READ exit key.

OFF disables Esc as a READ exit key.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

If SET ESCAPE is ON, Esc terminates the current READ. Any changes made to the current Get object are lost, and validation with RANGE or VALID is bypassed. When SET ESCAPE is OFF and the user presses Esc, the key press is ignored. With SET KEY, however, you can reassign Esc for special handling, regardless of the status of SET ESCAPE.

Files: Library is CLIPPER.LIB.

See Also: READ READEXIT() SET KEY SETCANCEL() SETKEY()

SET EXACT*
Toggle exact matches for character strings

Syntax

SET EXACT on | OFF | <xlToggle>

Arguments

ON enforces exact comparison of character strings, including length.

OFF resumes normal character string comparison.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET EXACT determines how two character strings are compared using the relational operators (=, >, <, =>, =<). When EXACT is OFF, strings are compared according to the following rules; assume two character strings cLeft and cRight where the expression to test is (cLeft = cRight):

. If cRight is a null string (""), return true (.T.).

. If LEN(cRight) is greater than LEN(cLeft), return false (.F.).

. Otherwise, compare all characters in cRight with cLeft. If all characters in cRight equal cLeft, return true (.T.); otherwise return false (.F.).

With EXACT ON, all relational operators except the double equal operator (==) treat two strings as equal, if they match exactly, excluding trailing spaces. With the double equal operator (==), all characters in the string are significant, including trailing spaces.

SET EXACT is a compatibility command and not recommended.

Notes

. **Compatibility:** In CA-Clipper, unlike other dialects, SET EXACT has no effect on operations other than relational operators. This includes the SEEK and FIND commands. If you need to seek exact matches of character keys, use the example user-defined function SeekExact() in the SEEK command reference.

Examples

. These examples show various results of the equal operator (=) with SET EXACT:

```
SET EXACT OFF
? "123" = "12345"            // Result: .F.

? "12345" = "123"            // Result: .T.
? "123" = ""                 // Result: .T.
? "" = "123"                 // Result: .F.
? "123" = "123  "            // Result: .F.
//
SET EXACT ON
? "123" = "12345"            // Result: .F.
? "12345" = "123"            // Result: .F.
? "123" = ""                 // Result: .F.
? "" = "123"                 // Result: .F.
? "123" = "123  "            // Result: .T.
```

Files: Library is CLIPPER.LIB.

See Also: SEEK

SET EXCLUSIVE*
Establish shared or exclusive USE of database files

Syntax

SET EXCLUSIVE ON | off | <xlToggle>

Arguments

ON causes database files to be opened in exclusive (nonshared) mode.

OFF causes database files to be opened in shared mode.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

In a network environment, SET EXCLUSIVE determines whether a USE command specified without the EXCLUSIVE or SHARED clause automatically opens database, memo, and index files EXCLUSIVE. When database files are opened EXCLUSIVE, other users cannot USE them until they are CLOSEd. In this mode, file and record locks are unnecessary.

When EXCLUSIVE is ON (the default), all database and associated files open in a nonshared (exclusive) mode unless the USE command is specified with the SHARED clause. Use EXCLUSIVE only for operations that absolutely require EXCLUSIVE USE of a database file, such as PACK, REINDEX, and ZAP.

When EXCLUSIVE is OFF, all files are open in shared mode unless the USE command is specified with the EXCLUSIVE clause. Control access by other users programmatically using RLOCK() and FLOCK().

SET EXCLUSIVE is a compatibility command and not recommended. It is superseded by the EXCLUSIVE and SHARED clauses of the USE command.

Refer to the Network Programming chapter in the Programming and Utilities guide for more information.

Notes

. **Error handling:** Attempting to USE a database file already opened EXCLUSIVE by another user generates a runtime error and sets NETERR() to true (.T.). After control returns to the point of error, you can test NETERR() to determine whether the USE failed.

Files: Library is CLIPPER.LIB.

See Also: FLOCK() NETERR() RLOCK() USE

SET FILTER
Hide records not meeting a condition

Syntax

SET FILTER TO [<lCondition>]

Arguments

TO <lCondition> is a logical expression that defines a specific set of current work area records accessible for processing.

SET FILTER TO without an argument deactivates the filter condition.

Description

When a FILTER condition is SET, the current work area acts as if it contains only the records that match the specified condition. A filter condition is one of the properties of a work area. Once a FILTER has been SET, the condition can be returned as a character string using the DBFILTER() function.

Most commands and functions that move the record pointer honor the current filter with the exception of those commands that access records by record number. This includes GOTO, commands specified with the RECORD clause, and RELATIONs linked by numeric expression to a work area with no active index.

Once a FILTER is SET, it is not activated until the record pointer is moved from its current position. You can use GO TOP to activate it.

As with SET DELETED, a filter has no effect on INDEX and REINDEX.

Note: Although SET FILTER makes the current work area appear as if it contains a subset of records, it, in fact, sequentially processes all records in the work area. Because of this, the time required to process a filtered work area will be the same as an unfiltered work area.

Examples

. This example filters Employee.dbf to only those records where the age is greater than 50:

```
USE Employee INDEX Name NEW
SET FILTER TO Age > 50
LIST Lastname, Firstname, Age, Phone
SET FILTER TO
```

Files: Library is CLIPPER.LIB.

See Also: DBFILTER() DBSETFILTER() SET DELETED

SET FIXED
Toggle fixing of the number of decimal digits displayed

Syntax

SET FIXED on | OFF | <xlToggle>

Arguments

ON fixes the decimal places display at the number of digits specified by SET DECIMALS.

OFF allows the operation or function to determine the number of decimal places to display.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET FIXED toggles control of the display of decimal digits by the current DECIMALS setting. When FIXED is ON, display of all numeric output is fixed at the DECIMALS setting (two places if the SET DECIMALS default value is in effect). When FIXED is OFF, numeric output displays according to the default rules for numeric display. These are described in the Basic Concepts chapter of the Programming and Utilities guide.

Note that SET FIXED and SET DECIMALS affect only the display format of numeric values and not the actual numeric precision of calculations.

Files: Library is CLIPPER.LIB.

See Also: EXP() LOG() SET DECIMALS SQRT() VAL()

SET FORMAT*
Activate a format when READ is executed

Syntax

SET FORMAT TO [<idProcedure>[.<ext>]]

Arguments

TO <idProcedure> is a format (.fmt) file, a program (.prg) file, or a procedure.

<ext> is the extension of the format file. If not specified, the default extension is (.fmt).

SET FORMAT TO with no argument deactivates the current format.

Description

SET FORMAT defines a procedure to execute when a READ is invoked. Unlike the interpreted environment, formats are not opened and executed at runtime. Instead, the CA-Clipper compiler treats SET FORMAT the same as a DO command. The compiler first looks to see whether it has already compiled a procedure with the same name as <idProcedure>. If it has, it uses that procedure for the reference. If <idProcedure> is not found, the compiler looks to disk for a file with the same name. If this file is not found, an external reference is generated that must be resolved at link time.

SET FORMAT is a compatibility command and not recommended.

Notes

- **Active format procedures:** Unlike other dialects where each work area can have an active format, CA-Clipper supports only one active format procedure for all work areas.

- **Screen CLEARing:** CA-Clipper does not clear the screen when a format procedure is executed.

- **Legal statements:** Format procedures allow statements and commands in addition to @...SAY and @...GET.

- **Multiple pages:** CA-Clipper does not support multiple-page format procedures.

Examples

- This example uses a format procedure to add records to a database file until the user presses Esc:

```
USE Sales NEW
SET FORMAT TO SalesScr
DO WHILE LASTKEY() != 27
   APPEND BLANK
   READ
ENDDO
RETURN

PROCEDURE SalesScr
   @ 12, 12 SAY "Branch     : " GET  Branch
   @ 13, 12 SAY "Salesman   : " GET  Salesman
   RETURN
```

Files: Library is CLIPPER.LIB.

See Also: @...GET @...SAY PROCEDURE READ

SET FUNCTION
Assign a character string to a function key

Syntax

SET FUNCTION <nFunctionKey> TO <cString>

Arguments

<nFunctionKey> is the number of the function key to receive the assignment.

TO <cString> specifies the character string to assign to <nFunctionKey>.

Description

SET FUNCTION assigns a character string to a function key numbered between 1 and 40 inclusive. When the user presses the assigned function key, <cString> is stuffed into the keyboard buffer. <cString> can include control characters, such as a Ctrl-C or Ctrl-S to perform navigation or editing actions in a wait state.

List of Function Key Mappings

Function Key	Actual Key
1 - 10	F1 - F10
11 - 20	Shift-F1 - Shift-F10
21 - 30	Ctrl-F1 - Ctrl-F10
31 - 40	Alt-F1 - Alt-F10

Warning! In CA-Clipper, SET FUNCTION is preprocessed into SET KEY and KEYBOARD commands. This means that SET FUNCTION has the effect of clearing any SET KEY for the same key number and vice versa. This is incompatible with previous releases, which maintained separate lists of SET FUNCTION keys and SET KEY keys.

Files: Library is CLIPPER.LIB.

See Also: KEYBOARD SET KEY SETKEY()

SET INDEX
Open index file(s) in the current work area

Syntax

SET INDEX TO [<xcIndex list>]

Arguments

TO <xcIndex list> is a list of up to 15 index (.ntx or .ndx) filenames, separated by commas. Any reference to an index filename that results in either a null string ("") or spaces is ignored. You can specify each index file as a literal filename or as a character expression enclosed in parentheses. If you specify no extension, a default extension is supplied by the current database driver. (.ntx) is the extension supplied by the CA-Clipper driver, and (.ndx) is the extension supplied by the dBASE III PLUS driver.

SET INDEX TO without an argument closes all indexes open in the current work area.

Description

SET INDEX closes all currently open index files then opens the specified index files in the current work area. When more than one index file is opened, the first index becomes the controlling index. The record pointer is initially positioned at the first logical record in the index. During database file processing, all open indexes are updated whenever a key value is appended or changed. To change the controlling index without issuing another SET INDEX command, use SET ORDER. To add indexes without closing the currently opened indexes, use DBSETINDEX().

SET INDEX is used primarily to open index files in a network environment instead of the INDEX clause of the USE command. Generally, USE the <xcDatabase> then test to determine whether the USE succeeded. If it did succeed, open the associated indexes with SET INDEX. See the example below.

Examples

. This example opens index files using extended expressions. Note how each index filename is specified by a separate variable:

```
xcIndex1:= "Name"
xcIndex2:= "Account"
SET INDEX TO (xcIndex1), (xcIndex2)
```

. This example demonstrates using SET INDEX in a network environment to open indexes:

```
USE Sales SHARED
IF !NETERR()
   SET INDEX TO Sales, Territory
ELSE
   ? "USE failed"
   BREAK
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: CLOSE DBSETINDEX() INDEX REINDEX SET ORDER USE

SET INTENSITY
Toggle enhanced display of GETs and PROMPTs

Syntax

SET INTENSITY ON | off | <xlToggle>

Arguments

ON enables both standard and enhanced display colors.

OFF disables enhanced display color. All screen output then uses the current standard color.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET INTENSITY toggles the display of GETs and menu PROMPTs between enhanced and standard color settings. When INTENSITY is OFF, GETs and SAYs appear in the standard color setting. When INTENSITY is ON, GETs appear in the enhanced color setting.

When INTENSITY is OFF, all menu PROMPTs appear in the standard color setting, and the cursor appears at the current PROMPT. If INTENSITY is ON (the default), the current PROMPT appears in the enhanced color setting, and the cursor is hidden.

Note that INTENSITY has no effect on ACHOICE() or DBEDIT().

Files: Library is CLIPPER.LIB.

See Also: @...GET @...PROMPT @...SAY SETCOLOR()

SET KEY
Assign a procedure invocation to a key

Syntax

SET KEY <nInkeyCode> TO [<idProcedure>]

Arguments

<nInkeyCode> is the INKEY() value of the key that receives the assignment.

TO <idProcedure> specifies the name of a procedure that executes when the user presses the assigned key. If <idProcedure> is not specified, the current <nInkeyCode> definition is released.

Description

SET KEY is a keyboard command that allows a procedure to be executed from any wait state when a designated key is pressed. A wait state is any mode that extracts keys from the keyboard except for INKEY(). These modes include ACHOICE(), DBEDIT(), MEMOEDIT(), ACCEPT, INPUT, READ and WAIT. After a key is redefined, pressing it executes the specified procedure, passing three automatic parameters corresponding to: PROCNAME(), PROCLINE(), and READVAR(). The procedure and variable parameters are character data type, while the line number is numeric data type.

You may define a maximum of 32 keys at one time. At startup, the system automatically defines the F1 key to execute Help. If a procedure with this name is linked into the current program and it is visible, pressing F1 from a wait state invokes it.

Note that SET KEY procedures should preserve the state of the application (i.e., screen appearance, current work area, etc.) and restore it before exiting.

Warning! In CA-Clipper, SET FUNCTION is preprocessed into the SET KEY and KEYBOARD commands. This means that SET FUNCTION has the effect of clearing any SET KEY for the same key number and vice versa. This is incompatible with previous releases, which maintained separate lists of SET FUNCTION keys and SET KEY keys.

Notes

- **Precedence:** SET KEY definitions take precedence over SET ESCAPE and SETCANCEL().
- **CLEAR with a SET KEY procedure:** Do not use CLEAR to clear the screen within a SET KEY procedure since it also CLEARs GETs and, therefore, terminates READ. When you need to clear the screen, use CLEAR SCREEN or CLS instead.
- **Terminating a READ from a SET KEY procedure:** The following table illustrates several ways to terminate a READ from within a SET KEY procedure.

Terminating a READ from a SET KEY Procedure

Command	Action
CLEAR GETS	Terminates READ without saving current GET
BREAK	Terminates READ without saving current GET
KEYBOARD Ctrl-W	Terminates READ and saves the current GET
KEYBOARD Esc	Terminates READ without saving current GET

Examples

. This example uses SET KEY to invoke a procedure that presents a picklist of account identification numbers when the user presses F2 while entering data into the account identification field:

```
#include   "Inkey.ch"

//
SET KEY K_F2 TO ScrollAccounts
USE Accounts NEW
USE Invoices NEW
@ 10, 10 GET Invoices->Id
READ
RETURN

PROCEDURE ScrollAccounts( cProc, nLine, cVar )
   IF cVar = "ID"
      SAVE SCREEN
      Accounts->(DBEDIT(10, 10, 18, 40, {"Company"}))
      KEYBOARD CHR(K_CTRL_Y) + Accounts->Id + ;
               CHR(K_HOME)
      RESTORE SCREEN
   ELSE
      TONE(100, 2)
   ENDIF
   RETURN
```

Files: Library is CLIPPER.LIB, header file is Inkey.ch.

See Also: INKEY() KEYBOARD READ SETKEY() Inkey codes

SET MARGIN
Set the page offset for all printed output

Syntax

SET MARGIN TO [<nPageOffset>]

Arguments

TO <nPageOffset> is a positive number that defines the number of column positions to indent from the left side of the page for subsequent printed output. A negative value resets the MARGIN to zero.

SET MARGIN TO with no argument resets the page offset to zero, the default value.

Description

SET MARGIN is valid for all output directed to the printer from console commands and @...SAY. With console output, the <nPageOffset> indent is output whenever there is a new line. With @...SAY, <nPageOffset> is added to each column value. SET MARGIN has no effect on screen output.

Note: Printing with @...SAY and PCOL() with a MARGIN SET in most cases adds the MARGIN to each column position. This happens since PCOL() accurately reflects the print column position including the last <nPageOffset> output. The best approach is to avoid the use of SET MARGIN with PCOL() for relative column addressing.

Examples

. This example sets a page offset of 5 then prints a list from Sales.dbf:

```
USE Sales NEW
SET MARGIN TO 5
LIST Branch, Salesman TO PRINTER
SET MARGIN TO
```

Files: Library is CLIPPER.LIB.

See Also: @...SAY PCOL() SET DEVICE SET PRINTER

SET MESSAGE
Set the @...PROMPT message line row

Syntax

SET MESSAGE TO [<nRow> [CENTER | CENTRE]]

Arguments

TO <nRow> specifies the message row position.

CENTER | CENTRE centers the message on the specified row.

Specifying SET MESSAGE TO 0 or SET MESSAGE TO without an argument suppresses the display of messages.

Description

SET MESSAGE is a menu command that defines the screen row where the @...PROMPT messages display. When a CA-Clipper program is invoked the default message row value is zero, suppressing all defined messages. Messages appear on <nRow>, column 0 unless the CENTER option is used.

Examples

. This example creates a small lightbar menu with an activated and centered message line:

```
SET MESSAGE TO 23 CENTER
SET WRAP ON
@ 5, 5 PROMPT "One" MESSAGE "Choice one"
@ 6, 5 PROMPT "Two" MESSAGE "Choice two"
MENU TO nChoice
//
IF nChoice == 0
   EXIT
ELSEIF nChoice == 1
   Proc1()
ELSEIF nChoice == 2
   Proc2()
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: @...PROMPT MENU TO SET WRAP

SET ORDER
Set a new controlling index

Syntax

SET ORDER TO [<nIndex>]

Arguments

TO <nIndex> specifies the new controlling index by pointing to its position in the list of open indexes in the current work area. This number can be in the range of zero to 15.

If <nIndex> is not specified, ORDER is SET to zero.

Description

When you SET ORDER TO a new controlling index, all indexes are properly updated when you either append or edit records. This is true even if you SET ORDER TO 0. After a change of controlling indexes, the record pointer still points to the same record.

SET ORDER TO 0 or SET ORDER TO with no argument restores the database file to natural order (record number order), leaving all indexes open.

To determine which index is the current controlling index, use INDEXORD().

Examples

. This example shows typical uses of the SET ORDER command:

```
USE Customers NEW
INDEX ON Lastname TO Names
INDEX ON City + State TO Region
SET INDEX TO Names, Region
//
SET ORDER TO 2
? INDEXKEY(INDEXORD())      // Result: City + State
//
SET ORDER TO 0
? INDEXKEY(INDEXORD())      // Result: null value
//
SET ORDER TO 1
? INDEXKEY(INDEXORD())      // Result: Lastname
```

Files: Library is CLIPPER.LIB.

See Also: INDEX INDEXORD() SEEK SET INDEX USE

SET PATH
Specify the CA-Clipper search path for opening files

Syntax

SET PATH TO [<xcPathspec list>]

Arguments

TO <xcPathspec list> identifies the paths CA-Clipper uses when searching for a file not found in the current directory. You can specify it as a literal list or as a character expression enclosed in parentheses. The list of paths can be separated by commas or semicolons. However, continuation of a SET PATH command line with a semicolon is not supported unless <xcPathspec> is specified as a character expression enclosed in parentheses.

Description

SET PATH allows commands and functions that open database and associated files to find and open existing files in another drive and/or directory. It does this by specifying a path list to search if a referenced file cannot be found in the DEFAULT or specified directory. Note that memo and low-level file functions respect neither the DEFAULT nor the PATH SETting.

A path is a pointer to a directory. It consists of an optional drive letter and colon, followed by a list of directories from the root to the desired directory separated by backslash (\) characters. A path list is the sequence of paths to search, each separated by a comma or semicolon.

When you attempt to access a file, CA-Clipper first searches the default drive and directory. The default disk drive and directory are established by DOS when your CA-Clipper application is loaded, or, during execution, by SET DEFAULT. If the file is not found, CA-Clipper then searches each path in the specified path list until the first occurrence of the file is found.

To create new files in another drive or directory, use SET DEFAULT TO <xcPathspec> or explicitly declare the path when specifying a new filename.

SET PATH TO with no argument releases the path list and CA-Clipper searches only the DEFAULT directory.

Examples

. This example is a typical PATH command:

```
SET PATH TO A:\INVENTORY;B:\VENDORS
```

. This example configures a path setting at runtime by passing a DOS environment variable to a program, retrieving its value with GETENV(), then SETting PATH with this value. For example, in DOS:

```
SET CLIP_PATH=C:\APPS\DATA,C:\APPS\PROGS
```

Later in the configuration section of your application program:

```
SET PATH TO (GETENV("CLIP_PATH"))
```

Files: Library is CLIPPER.LIB.

See Also: CURDIR() SET DEFAULT

AFILL()
Fill an array with a specified value

Syntax

AFILL(<aTarget>, <expValue>,
[<nStart>], [<nCount>]) --> aTarget

Arguments

<aTarget> is the array to fill.

<expValue> is the value to place in each array element. It can be an expression of any valid data type.

<nStart> is the position of the first element to fill. If this argument is omitted, the default value is one.

<nCount> is the number of elements to fill starting with element <nStart>. If this argument is omitted, elements are filled from the starting element position to the end of the array.

Returns

AFILL() returns a reference to <aTarget>.

Description

AFILL() is an array function that fills the specified array with a single value of any data type (including an array, code block, or NIL) by assigning <expValue> to each array element in the specified range.

Warning! AFILL() cannot be used to fill multidimensional arrays. CA-Clipper implements multidimensional arrays by nesting arrays within other arrays. Using AFILL() with a multidimensional array will overwrite subarrays used for the other dimensions of the array.

Examples

. This example, creates a three-element array. The array is then filled with the logical value, (.F.). Finally, elements in positions two and three are assigned the new value of true (.T.):

```
LOCAL aLogic[3]
// Result: aLogic is { NIL, NIL, NIL }

AFILL(aLogic, .F.)
// Result: aLogic is { .F., .F., .F. }

AFILL(aLogic, .T., 2, 2)
// Result: aLogic is { .F., .T., .T. }
```

Files: Library is CLIPPER.LIB.

See Also: AADD() AEVAL() DBSTRUCT() DIRECTORY()

SET PRINTER
Toggle echo of console output to the printer or set the destination of printed output

---

Syntax

SET PRINTER on | OFF | <xlToggle>
SET PRINTER TO [<xcDevice> | <xcFile> [ADDITIVE]]

Arguments

ON echoes console output to the printer.

OFF suppresses the printing of console output.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

TO <xcDevice> identifies the name of the device where all subsequent printed output will be sent. You can specify a device name as a literal character string or a character expression enclosed in parentheses. Additionally, a device can be either local or network. SETting PRINTER TO a nonexisting device creates a file with the name of the device. When specifying device names, do not use a trailing colon.

TO <xcFile> identifies the name of the output file. You can specify the filename as a literal string or as a character expression enclosed in parentheses. If a file extension is not specified, (.prn) is assumed.

ADDITIVE causes the specified output file to be appended to instead of overwritten. If ADDITIVE is not specified, an existing output file is truncated before new information is written to it. The ADDITIVE clause is only meaningful when SETting PRINTER TO an output file.

If SET PRINTER TO is specified with no arguments, the currently specified device or file is closed and the default destination is then reselected.

Description

SET PRINTER, like many other SET commands, has two basic forms with each having its own functionality. The on|OFF form of SET PRINTER controls whether the output of console commands is echoed to the printer. Console commands generally do not specify row and column coordinates. All of these commands, except ?|??, have a TO PRINTER clause that also directs output to the printer. Output from console commands is displayed to the screen unless CONSOLE is OFF. Be aware that @...SAYs are not affected by SET PRINTER ON. To send them to the printer, use SET DEVICE TO PRINTER instead.

SET PRINTER TO determines the destination of output from all commands and functions that send output to the printer. This includes @...SAYs if DEVICE is SET TO PRINTER. Output can be sent to a device or to a file. If the destination is a device, the following names are valid: LPT1, LPT2, LPT3 (all parallel ports), COM1, and COM2 (serial ports), CON and PRN. The default device is PRN.

If the destination is a file, it is created in the current DEFAULT directory. If a file with the same name exists in the same location, it is overwritten by the new file without warning. All subsequent output to the printer is then written to this file until the file is closed using SET PRINTER TO with no argument.

Use SET PRINTER TO for:

- Managing multiple printers by swapping ports

- Directing output to a file for printing later or for transferring to a remote computer via telecommunications

- Emptying the printer spooler and resetting the default device

Notes

- **Compatibility:** CA-Clipper does not support the syntax SET PRINTER TO \\SPOOLER or \\CAPTURE. Specifying SET PRINTER with either of these options creates the files Spooler.prn or Capture.prn. The symbols \\ are ignored.

- **End of file marks:** When printer output is redirected to a file, an end of file mark (CHR(26)) is not written when the file is closed. To terminate a file with an end of file mark, issue a ?? CHR(26) just before the SET PRINTER command that closes the file.

- **Networking:** For some networks, the workstation's printer should first be redirected to the file server (usually by running the network spooler program).

Examples

- This example echoes the output of the ? command to printer, suppressing the console screen display by SETting CONSOLE OFF:

```
USE Customer NEW
SET PRINTER ON
SET CONSOLE OFF
DO WHILE !EOF()

   ? Customer->Name, Customer->Phone
   SKIP
ENDDO
EJECT
SET PRINTER OFF
SET CONSOLE ON
CLOSE
RETURN
```

- This example directs printer output to LPT1 and empties the print spooler upon completion:

```
SET PRINTER TO LPT1
<Printing statements>...
SET PRINTER TO            // Empty the print spooler
```

- This example sends printer output to a text file, overwriting an existing file with the same name:

```
SET PRINTER TO Prnfile.txt
SET DEVICE TO PRINTER
SET PRINTER ON
//
@ 0, 0 SAY "This goes to Prnfile.txt"
? "So will this!"
//
SET DEVICE TO SCREEN
SET PRINTER OFF
SET PRINTER TO            // Close the print file
```

Files: Library is CLIPPER.LIB.

See Also: @...SAY EJECT SET CONSOLE SET DEVICE SETPRC()

SET PROCEDURE*
Compile procedures and functions into the current .OBJ file

Syntax

SET PROCEDURE TO [<idProgramFile>[.<ext>]]

Arguments

TO <idProgramFile> is the name of the procedure file to compile into the current object file. It can optionally include a path and/or drive designator.

<ext> is the optional extension of the procedure. If not specified, (.prg) is assumed.

SET PROCEDURE TO with no argument is ignored.

Description

SET PROCEDURE directs the compiler to compile all procedures and user-defined functions declared within the specified procedure file into the current object file (.OBJ).

SET PROCEDURE is a compatibility command and not recommended. It has been superseded by other facilities more appropriate to the compiled environment (e.g., the compiler script file (.clp)).

See the CA-Clipper Compiler chapter in the Programming and Utilities guide for a full discussion of program architecture and configuration.

See Also: #include DO* FUNCTION PROCEDURE RETURN

SET RELATION
Relate two work areas by a key value or record number

Syntax

```
SET RELATION TO [<expKey> | <nRecord> INTO <xcAlias>]
   [, [TO] <expKey2> | <nRecord2> INTO <xcAlias2>...]
   [ADDITIVE]
```

Arguments

TO <expKey> is an expression that performs a SEEK in the child work area each time the record pointer moves in the parent work area. For this to work, the child work area must have an index in USE.

TO <nRecord> is an expression that performs a GOTO to the matching record number in the child work area each time the record pointer moves in the parent work area. If <nRecord> evaluates to RECNO(), the relation uses the parent record number to perform a GOTO to the same record number in the child work area. For a numeric expression type of relation to execute correctly, the child work area must not have an index in USE.

INTO <xcAlias> identifies the child work area and can be specified either as the literal alias name or as a character expression enclosed in parentheses.

ADDITIVE adds the specified child relations to existing relations already set in the current work area. If this clause is not specified, existing relations in the current work area are released before the new child relations are set.

SET RELATION TO with no arguments releases all relations defined in the current work area.

Description

SET RELATION is a database command that links a parent work area to one or more child work areas using a key expression, record number, or numeric expression. Each parent work area can be linked to as many as eight child work areas. A relation causes the record pointer to move in the child work area in accordance with the movement of the record pointer in the parent work area. If no match is found in the child work area, the child record pointer is positioned to LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.).

The method of linking the parent and child work areas depends on the type of <expKey> and presence of an active index in the child work area. If the child work area has an active index, the lookup is a standard SEEK. If the child work area does not have an active index and the type of <expKey> is numeric, a GOTO is performed in the child work area instead.

Notes

- **Cyclical relations:** Do not relate a parent work area to itself either directly or indirectly.
- **Soft seeking:** SET RELATION does not support SOFTSEEK and always behaves as if SOFTSEEK is OFF even if SOFTSEEK is ON. This means that if a match is not found in the child work area, the child record pointer is always positioned to LASTREC() + 1.
- **Record number relations:** To relate two work areas based on matching record numbers, use RECNO() for the SET RELATION TO expression and make sure the child work area has no active indexes.

Examples

. This example relates three work areas in a multiple parent-child configuration with Customer related to both Invoices and Zip:

```
USE Invoices INDEX Invoices NEW
USE Zip INDEX Zipcode NEW
USE Customer NEW
SET RELATION TO CustNum INTO Invoices, Zipcode INTO Zip
LIST Customer, Zip->City, Invoices->Number, ;
         Invoices->Amount
```

. Sometime later, you can add a new child relation using the ADDITIVE clause, like this:

```
USE BackOrder INDEX BackOrder NEW
SELECT Customer
SET RELATION TO CustNum INTO BackOrder ADDITIVE
```

Files: Library is CLIPPER.LIB.

See Also: DBRELATION() DBRSELECT() FOUND() RECNO()

SET SCOREBOARD
Toggle the message display from READ or MEMOEDIT()

Syntax

SET SCOREBOARD ON | off | <xlToggle>

Arguments

ON allows the display of messages from READ and MEMOEDIT() on line zero of the screen.

OFF suppresses these messages.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET SCOREBOARD controls whether or not messages from READ and MEMOEDIT() display on line zero. When SCOREBOARD is ON, READ displays messages for RANGE errors, invalid dates, and insert status. MEMOEDIT() displays an abort query message and the insert status.

To suppress the automatic display of these messages, SET SCOREBOARD OFF.

Files: Library is CLIPPER.LIB.

See Also: @...GET MEMOEDIT() READ

SET SOFTSEEK
Toggle relative SEEKing

Syntax

SET SOFTSEEK on | OFF | <xlToggle>

Arguments

ON causes the record pointer to be moved to the next record with a higher key after a failed index search.

OFF causes the record pointer to be moved to EOF() after a failed index search.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET SOFTSEEK enables relative seeking, a method of searching an index and returning a record even if there is no match for a specified key.

When SOFTSEEK is ON and a match for a SEEK is not found, the record pointer is set to the next record in the index with a higher key value than the SEEK argument. Records not visible because SET FILTER and/or SET DELETED are skipped when searching for the next higher key value. If there is no record with a higher key value, the record pointer is positioned at LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.). FOUND() returns true (.T.) only if the record is actually found. It never returns true (.T.) for a relative find.

When SOFTSEEK is OFF and a SEEK is unsuccessful, the record pointer is positioned at LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.).

Notes

. **SET RELATION:** SET RELATION ignores SOFTSEEK updating the record pointer in all linked child work areas as if SOFTSEEK is OFF.

Examples

. This example illustrates the possible results of a SEEK with SET SOFTSEEK ON:

```
SET SOFTSEEK ON
USE Salesman INDEX Salesman NEW
ACCEPT "Enter Salesman: " TO cSearch
SEEK cSearch
DO CASE
CASE FIELD->Salesman = cSearch
   ? "Match found:", FOUND(), EOF(), FIELD->Salesman
CASE !EOF()
   ? "Soft match found:", FOUND(), EOF(), ;
             FIELD->Salesman
OTHERWISE
   ? "No key matches:", FOUND(), EOF(), FIELD->Salesman
ENDCASE
```

Files: Library is CLIPPER.LIB.

See Also: FOUND() SEEK SET INDEX SET ORDER SET RELATION

SET TYPEAHEAD
Set the size of the keyboard buffer

Syntax

SET TYPEAHEAD TO <nKeyboardSize>

Arguments

TO <nKeyboardSize> specifies the number of keystrokes the keyboard buffer can hold from a minimum of zero to a maximum of 4096. The default size of the keyboard buffer is machine-dependent but 16 is the minimum size.

Description

SET TYPEAHEAD defines the size of the CA-Clipper keyboard buffer that caches keystrokes input directly by the user. SET TYPEAHEAD, however, does not affect the number of characters that can be stuffed programmatically using the KEYBOARD command. When executed, SET TYPEAHEAD clears the keyboard buffer and sets the size to <nKeyboardSize>.

When TYPEAHEAD is SET TO zero, keyboard polling is suspended. An explicit request for keyboard input, however, will temporarily enable the keyboard and read any pending keystrokes from the BIOS buffer. Calling NEXTKEY() constitutes such an explicit request. NEXTKEY() reads any pending keystrokes from the BIOS buffer and returns the INKEY() value of the first keystroke read, or zero if no keystrokes are pending.

Files: Library is CLIPPER.LIB.

See Also: ALTD() CLEAR TYPEAHEAD INKEY() KEYBOARD NEXTKEY()

SET UNIQUE*
Toggle inclusion of nonunique keys into an index

Syntax

SET UNIQUE on | OFF | <xlToggle>

Arguments

ON causes index files to be created with a uniqueness attribute.

OFF causes index files to be created without a uniqueness attribute.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET UNIQUE is a database command that controls whether indexes are created with uniqueness as an attribute. With UNIQUE ON, new indexes are created including only unique keys. This is the same as creating an index with the INDEX...UNIQUE command.

If, during the creation or update of an unique index, two or more records are encountered with the same key value, only the first record is included in the index. When the unique index is updated, REINDEXed, or PACKed, only unique records are maintained, without regard to the current SET UNIQUE value.

Changing key values in a unique index has important implications. First, if a unique key is changed to the value of a key already in the index, the changed record is lost from the index. Second, if there is more than one instance of a key value in a database file, changing the visible key value does not bring forward another record with the same key until the index is rebuilt with REINDEX, PACK, or INDEX...UNIQUE.

With UNIQUE OFF, indexes are created with all records in the index. Subsequent updates to the database files add all key values to the index independent of the current UNIQUE SETting.

SET UNIQUE is a compatibility command not recommended. It is superseded by the UNIQUE clause of the INDEX command.

Files: Library is CLIPPER.LIB.

See Also: DBCREATEIND() INDEX PACK REINDEX SEEK

SET WRAP
Toggle wrapping of the highlight in MENUs

Syntax

SET WRAP on | OFF | <xlToggle>

Arguments

ON enables the highlight to wrap around when navigating a lightbar menu.

OFF disables wrapping when navigating a lightbar menu.

<xlToggle> is a logical expression that must be enclosed in parentheses. A value of true (.T.) is the same as ON, and a value of false (.F.) is the same as OFF.

Description

SET WRAP is a menu command that toggles wrapping of the highlight in an @...PROMPT menu from the first menu item to the last menu item and vice versa. When WRAP is ON and the last menu item is highlighted, Rightarrow or Dnarrow moves the highlight to the first menu item. Likewise, when the first menu item is highlighted, Leftarrow or Uparrow moves the highlight to the last menu item.

When WRAP is OFF, pressing Uparrow or Leftarrow from the first menu item or Dnarrow or Rightarrow from the last menu item does nothing.

Files: Library is CLIPPER.LIB.

See Also: @...PROMPT MENU TO SET MESSAGE

SKIP
Move the record pointer to a new position

Syntax

SKIP [<nRecords>] [ALIAS <idAlias> | <nWorkArea>]

Arguments

<nRecords> is a numeric expression specifying the number of records to move the record pointer from the current position. A positive value moves the record pointer forward and a negative value moves the record pointer backward.

ALIAS <idAlias>|<nWorkArea> specifies the alias name as a literal identifier or the work area as a numeric expression.

SKIP specified with no arguments moves the record pointer forward one record.

Description

SKIP moves the record pointer to a new position relative to the current position in the current work area and within the current filter if there is one. SKIP is generally used for operations, such as reporting, that need to go to the next record in a database file.

If the alias clause is specified, the pointer can be moved in another work area without SELECTing that work area. SKIP can move either forward or backward. If there is no active index, SKIP moves the record pointer relative to the current position in the target database file. If there is an active index, SKIP moves the pointer relative to the current position in the index instead of the database file.

Attempting to SKIP forward beyond the end of file positions the record pointer at LASTREC() + 1 and EOF() returns true (.T.). Attempting to SKIP backward beyond the beginning of file moves the pointer to the first record and BOF() returns true (.T.).

In a network environment, any record movement command, including SKIP, makes changes to the current work area visible to other applications if the current file is shared and the changes were made during an RLOCK(). To force an update to become visible without changing the current record position, use SKIP 0. If, however, the changes were made during an FLOCK(), visibility is not guaranteed until the lock is released, a COMMIT is performed, or the file is closed. Refer to the Network Programming chapter in the Programming and Utilities guide for more information.

Examples

. This example uses SKIP with various arguments and shows the result:

```
USE Customers NEW
SKIP
? RECNO()                  // Result: 2
SKIP 10
? RECNO()                  // Result: 12
SKIP -5
? RECNO()                  // Result: 7
```

. This example moves the record pointer in a remote work area:

```
USE Customers NEW

USE Invoices NEW
```

```
SKIP ALIAS Customers
```

. This example prints a report using SKIP to move the record pointer sequentially through the Customer database file:

```
LOCAL nLine := 99
USE Customers NEW
SET PRINTER ON
DO WHILE !EOF()
   IF nLine > 55
      EJECT
      nLine := 1
   ENDIF
      Customer, Address, City, State, Zip
      nLine++
      SKIP
   ENDDO
SET PRINTER OFF
CLOSE Customers
```

Files: Library is CLIPPER.LIB.

See Also: BOF() COMMIT DBSKIP() EOF() GO LOCATE RECNO() SEEK

SORT
Copy to a (.dbf) file in sorted order

Syntax

```
SORT TO <xcDatabase> ON <idField> [/[A | D][C]]
   [, <idField2> [/[A | D][C]]...]
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
```

Arguments

TO <xcDatabase> is the name of the target file for the sorted records and can be specified either as a literal filename or as a character expression enclosed in parentheses. Unless otherwise specified, the new file is assigned a (.dbf) extension.

ON <idField> is the sort key and must be a field variable.

/[A|D][C] specifies how <xcDatabase> is to be sorted. /A sorts in ascending order. /D sorts in descending order. /C sorts in dictionary order by ignoring the case of the specified character field. The default SORT order is ascending.

<scope> is the portion of the current database file to SORT. The default is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to SORT within the given scope.

Description

SORT is a database command that copies records from the current work area to another database file in sorted order. CA-Clipper SORTs character fields in accordance with the ASCII value of each character within the string unless the /C option is specified. This option causes the database file to be SORTed in dictionary order--capitalization is ignored. Numeric fields are SORTed in numeric order, date fields are SORTed chronologically, and logical fields are sorted with true (.T.) as the high value. Memo fields cannot be SORTed.

SORT performs as much of its operation as possible in memory then it spools to a uniquely named temporary disk file. This temporary file can be as large as the size of the source database file. Note also that a SORT uses up three file handles: the source database file, the target database file, and the temporary file.

In a network environment, you must lock the database file to be SORTed with FLOCK() or USE it EXCLUSIVEly.

Notes

. **Deleted source records:** If DELETED is OFF, SORT copies deleted records to the target database file; however, the deleted records do not retain their deleted status. No record is marked for deletion in the target file regardless of its status in the source file.

  If DELETED is ON, deleted records are not copied to the target database file. Similarly, filtered records are ignored during a SORT and are not included in the target file.

. **SET EXACT:** SET EXACT has no effect on SORTing.

Examples

. This example copies a sorted subset of a mailing list to a smaller list for printing:

```
USE Mailing INDEX Zip
SEEK "900"
SORT ON LastName, FirstName TO Invite WHILE Zip = "900"
USE Invite NEW
REPORT FORM RsvpList TO PRINTER
```

Files: Library is CLIPPER.LIB.

See Also: ASORT() FLOCK() INDEX USE

STORE*
Assign a value to one or more variables

Syntax

STORE <exp> TO <idVar list>
<idVar> = <exp>
<idVar> := [<idVar2> := ...] <exp>

Arguments

<exp> is a value of any data type that is assigned to the specified variables.

TO <idVar list> defines a list of one or more local, static, public, private, or field variables that are assigned the value <exp>. If any <idVar> is not visible or does not exist, a private variable is created and assigned <exp>.

Description

STORE assigns a value to one or more variables of any storage class. The storage classes of CA-Clipper variables are local, static, field, private, and public. STORE is identical to the simple assignment operators (=) and (:=). In fact, a STORE statement is preprocessed into an assignment statement using the inline operator (:=). Like all of the assignment operators, STORE assigns to the most recently declared and visible variable referenced by <idVar>. If, however, the variable reference is ambiguous (i.e., not declared at compile time or not explicitly qualified with an alias), it is assumed to be MEMVAR. At runtime, if no private or public variable exists with the specified name, a private variable is created.

To override a declaration, you can specify the <idVar> prefaced by an alias. If <idVar> is a field variable, use the name of the work area. For private and public variables, you can use the memory variable alias (MEMVAR->). To assign to a field variable in the currently selected work area (as opposed to a particular named work area), you can use the field alias (FIELD->).

As a matter of principle, all variables other than field variables should be declared. Preface field variables with the alias. Use of private and public variables is discouraged since they violate basic principles of modular programming and are much slower than local and static variables.

Note that the STORE command is a compatibility command and not recommended for any assignment operation. CA-Clipper provides assignment operators that supersede the STORE command, including the inline assignment operator (:=), the increment and decrement operators (++) and (--), and the compound assignment operators (+=, -=, *=, /=). Refer to the Operators and Variables sections of the Basic Concepts chapter in the Programming and Utilities guide for more information.

Notes

. **Assigning a value to an entire array:** In CA-Clipper, neither the STORE command nor the assignment operators can assign a single value to an entire array. Use the AFILL() function for this purpose.

. **Memo fields:** Assigning a memo field to a variable assigns a character value to that variable.

Examples

. These statements create and assign values to undeclared private variables:

```
STORE "string" TO cVar1, cVar2, cVar3
cVar1:= "string2"
cVar2:= MEMVAR->cVar1
```

. These statements assign multiple variables using both STORE and the inline assignment operator (:=). The methods produce identical code:

```
STORE "value" to cVar1, cVar2, cVar3
cVar1 := cVar2 := cVar3 := "value"
```

. These statements assign values to the same field referenced explicitly with an alias. The first assignment uses the field alias (FIELD->), where the second uses the actual alias name:

```
USE Sales NEW
FIELD->CustBal = 1200.98
Sales->CustBal = 1200.98
```

Files: Library is CLIPPER.LIB.

See Also: PRIVATE PUBLIC RELEASE REPLACE RESTORE SAVE

AINS()
Insert a NIL element into an array

Syntax

AINS(<aTarget>, <nPosition>) --> aTarget

Arguments

<aTarget> is the array into which a new element will be inserted.

<nPosition> is the position at which the new element will be inserted.

Returns

AINS() returns a reference to the target array, <aTarget>.

Description

AINS() is an array function that inserts a new element into a specified array. The newly inserted element is NIL data type until a new value is assigned to it. After the insertion, the last element in the array is discarded, and all elements after the new element are shifted down one position.

Warning! AINS() must be used carefully with multidimensional arrays. Multidimensional arrays in CA-Clipper are implemented by nesting arrays within other arrays. Using AINS() in a multidimensional array discards the last element in the specified target array which, if it is an array element, will cause one or more dimensions to be lost. To insert a new dimension into an array, first add a new element to the end of the array using AADD() or ASIZE() before using AINS().

Examples

. This example demonstrates the effect of using AINS() on an array:

```
LOCAL aArray
aArray := { 1, 2, 3 }      // Result: aArray is
                           // now { 1, 2, 3 }
AINS(aArray, 2)            // Result: aArray is
                           // now { 1, NIL, 2 }
```

Files: Library is CLIPPER.LIB.

See Also: AADD() ACOPY() ADEL() AEVAL() AFILL() ASIZE()

SUM
Sum numeric expressions to variables

Syntax

SUM <nExp list> TO <idVar list>
[<scope>] [WHILE <lCondition>] [FOR <lCondition>]

Arguments

<nExp list> is the list of numeric values to sum for each record processed.

TO <idVar list> identifies the receiving variables to assign the results of the sum. Variables that either do not exist or are not visible are created as private variables. <idVar list> must contain the same number of elements as <nExp list>.

<scope> is the portion of the current database file to SUM. The default scope is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to SUM within the given scope.

Description

SUM is a database command that totals a series of numeric expressions for a range of records in the current work area and assigns the results to a series of variables. The variables specified in <idVar list> can be field, local, private, public, or static.

Note that the <nExp list> is required and not optional as it is in other dialects.

Examples

. This example illustrates the use of SUM:

```
LOCAL nTotalPrice, nTotalAmount
USE Sales NEW
SUM Price * .10, Amount TO nTotalPrice, nTotalAmount
//
? nTotalPrice               // Result: 151515.00
? nTotalAmount              // Result: 150675.00
```

Files: Library is CLIPPER.LIB.

See Also: AVERAGE DBEVAL() TOTAL

TEXT*
Display a literal block of text

Syntax

```
TEXT [TO PRINTER] [TO FILE <xcFile>]
   <text>...
ENDTEXT
```

Arguments

<text> is the block of literal characters to display to the screen. Text is displayed exactly as formatted.

TO PRINTER echoes the display to the printer.

TO FILE <xcFile> echoes the display to the specified file. <xcFile> may be specified as a literal filename or as a character expression enclosed in parentheses. If no extension is specified, (.txt) is assumed.

Description

TEXT...ENDTEXT is a console command construct that displays a block of text to the screen, optionally echoing output to the printer and/or a text file. To suppress output to the screen while printing or echoing output to a file, SET CONSOLE OFF before the TEXT command line.

Text within the TEXT construct displays exactly as formatted, including any indentation. Hard carriage returns are output as new lines, soft carriage returns as the character CHR(141). Macro variables found within TEXT...ENDTEXT are expanded. However, macro expressions are not.

TEXT...ENDTEXT is a compatibility command and not recommended. CA-Clipper has other facilities for text processing and output. For example, MEMOLINE() in combination with MLCOUNT() can word wrap long strings according to a specified line length. ? or @...SAY can display formatted text extracted from a long string with MEMOLINE().

Examples

. This example demonstrates how to use TEXT...ENDTEXT to print a form letter:

```
USE Sales NEW
DO WHILE !EOF()
   FormLetter()
   SKIP
ENDDO
RETURN

FUNCTION FormLetter
   LOCAL dDate := DTOC(DATE()), cSalesman := ;
            RTRIM(Salesman)
   TEXT TO PRINTER
   &dDate.
   Dear &cSalesman.,
   How are you!
   ENDTEXT
   EJECT
   RETURN NIL
```

Files: Library is CLIPPER.LIB.

See Also: ?|?? @...SAY MEMOLINE() MLCOUNT() SET CONSOLE

TOTAL
Summarize records by key value to a (.dbf) file

Syntax

```
TOTAL ON <expKey> FIELDS <idField list>
   TO <xcDatabase>
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
```

Arguments

ON <expKey> defines the group of records that produce a new record in the target database file. To make the summarizing operation accurate, the source database file should be INDEXed or SORTed on this expression.

FIELDS <idField list> specifies the list of numeric fields to TOTAL. If the FIELDS clause is not specified, no numeric fields are totaled. Instead each numeric field in the target file contains the value for the first record matching the key expression.

TO <xcDatabase> is the name of the target file that will contain the copy of the summarized records. Specify this argument as a literal filename or as a character expression enclosed in parentheses. Unless otherwise specified, TOTAL assumes a (.dbf) extension.

<scope> is the portion of the current database file to TOTAL. The default is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to TOTAL within the given scope.

Description

TOTAL is a database command that sequentially processes the current database file summarizing records by the specified key value and copying them to a new database file. TOTAL works by first copying the structure of the current database file to <xcDatabase>, except for memo fields. It then sequentially scans the current database file within the specified scope of records. As each record with a unique <expKey> value is encountered, that record is copied to the new database file. The values of numeric fields specified in <idField list> from successive records with the same <expKey> value are added to fields with the same names in <xcDatabase>. Summarization proceeds until a record with a new key value is encountered. The process is then repeated for this record.

Since TOTAL processes the source database file sequentially, it must be INDEXed or SORTed in <expKey> order for the summarization to be correct.

To successfully TOTAL numeric fields, the source numeric fields must be large enough to hold the largest total possible for that numeric field. If not, a runtime error is generated.

Notes

. **Deleted source records:** If DELETED is OFF, deleted records in the source file are TOTALed. Records in the target <xcDatabase> inherit the deleted status of the first matching record in the source file, just as nontotaled fields inherit their values. If DELETED is ON, however, none of the deleted source records are TOTALed.

Examples

. In this example, a database file is TOTALed ON the key

expression of the controlling index using a macro expression. When the macro expression is encountered, the expression is evaluated and the resulting character string is substituted for the TOTAL <expKey> argument:

```
USE Sales INDEX Branch NEW
TOTAL ON &(INDEXKEY(0)) FIELDS Amount TO Summary
```

Files: Library is CLIPPER.LIB.

See Also: AVERAGE INDEX SORT SUM

TYPE
Display the contents of a text file

Syntax

TYPE <xcFile> [TO PRINTER] [TO FILE <xcOutFile>]

Arguments

<xcFile> is the name of the file, including extension, to display to the screen. This argument may be specified as a literal filename or as a character expression enclosed in parentheses. <xcFile> must be specified with an extension if it has one.

TO PRINTER echoes the display to the printer.

TO FILE <xcOutFile> echoes the display to the specified file. <xcOutFile> may be specified either as a literal filename or as a character expression enclosed in parentheses. If no extension is specified, (.txt) is added.

Description

TYPE is a console command that displays the contents of a text file to the screen, optionally echoing the display to the printer and/or another text file. To suppress output to the screen while printing or echoing output to a file, SET CONSOLE OFF before the TYPE invocation.

If <xcFile> is specified without a path and/or drive designator, TYPE searches the current DEFAULT directory then the current PATH. If <xcOutFile> is specified without a path and/or drive designator, TYPE creates the file in the current DEFAULT directory.

TYPE performs no special formatting on the listing. There are no special headings or pagination when the output is sent to the printer.

To pause output, use Ctrl-S. Note that you cannot interrupt a listing with Esc.

Examples

. This example illustrates the TYPE command:

```
TYPE Main.prg TO PRINTER
```

Files: Library is EXTEND.LIB.

See Also: COPY FILE SET DEFAULT SET PATH SET PRINTER

UNLOCK
Release file/record locks set by the current user

Syntax

UNLOCK [ALL]

Arguments

ALL releases all current locks in all work areas. If not specified, only the lock in the current work area is released.

Description

UNLOCK is a network command that releases file or record locks set by the current user. Use it when you want to release the current lock without setting a new lock. Both FLOCK() and RLOCK() release the current lock before setting a new one.

After an UNLOCK, an update to a shared database file and associated index and memo files becomes visible to DOS and other applications but is not guaranteed to appear on disk until you perform a COMMIT or close the file.

Refer to the Network Programming chapter in the Programming and Utilities guide for more information on the principles of locking and update visibility.

Notes

. **SET RELATION:** UNLOCK does not automatically release a record lock along a RELATION chain unless you UNLOCK ALL.

Examples

. This example attempts an update operation that requires a record lock. If the RLOCK() is successful, the record is updated with a user-defined function and the RLOCK() is released with UNLOCK:

```
USE Sales INDEX Salesman SHARED NEW
IF RLOCK()
   UpdateRecord()
   UNLOCK
ELSE
   ? "Record update failed"
   BREAK
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: FLOCK() RLOCK() SET RELATION USE DBUNLOCK()

UPDATE
Update current database file from another database file

Syntax

```
UPDATE FROM <xcAlias>
   ON <expKey> [RANDOM]
   REPLACE <idField> WITH <exp>
   [, <idField2> WITH    <exp2>...]
```

Arguments

FROM <xcAlias> specifies the alias of the work area used to update records in the current work area. This argument may be specified either as a literal filename or as a character expression enclosed in parentheses.

ON <expKey> specifies the expression that defines matching records in the FROM work area.

REPLACE <idField> specifies a field in the current work area to replace with a new value.

WITH <exp> specifies the value to replace into the current field. You must reference any field contained in the FROM work area with the correct alias.

RANDOM allows records in the FROM database file to be in any order. If this option is specified, the current database file must be indexed on <expKey>.

Description

UPDATE is a database command that replaces fields in the current work area with values from another work area based on the specified key expression. UPDATE is designed to update only current work area records based on a one-to-one or one-to-many relation with the FROM work area. This means that UPDATE can only update records in the current work area with unique key values. When there is more than one instance of a key value, only the first record with the key value is updated. The FROM work area, however, can have duplicate key values.

There are two formulations of the command depending on whether the FROM work area records are sorted or indexed on <expKey> or not. If RANDOM is not specified, both the current work area and the FROM work area must be indexed or sorted in <expKey> order. If RANDOM is specified, the current work area must be indexed by <expKey>, but the FROM work area records can be in any order.

To use UPDATE in a network environment, the current database file must be locked with FLOCK() or USEed EXCLUSIVEly. The FROM database file may be used in any mode. Refer to the Network Programming chapter in the Programming and Utilities guide for more information.

Notes

. **Deleted records:** If DELETED is OFF, deleted records in both source files are processed. Records in the file being updated retain their deleted status and are not affected by the deleted status of records in the FROM file. If DELETED is ON, however, no deleted records are processed from either source file.

Examples

. This example UPDATEs Customers database file with outstanding invoice amounts:

```
USE Invoices NEW
USE Customer INDEX Customer NEW
UPDATE FROM Invoices ON Last;
   REPLACE Owed WITH Owed + Invoices->Amount RANDOM
```

Files: Library is CLIPPER.LIB.

See Also: INDEX JOIN REPLACE SET UNIQUE* SORT TOTAL

USE
Open an existing (.dbf) and its associated files

Syntax

```
USE [<xcDatabase>
   [INDEX <xcIndex list>]
   [ALIAS <xcAlias>] [EXCLUSIVE | SHARED]
   [NEW] [READONLY]
   [VIA <cDriver>]]
```

Arguments

<xcDatabase> is the name of the database file to open and may be specified either as a literal filename or as a character expression enclosed in parentheses.

INDEX <xcIndex list> specifies the names of 1 to 15 index files to open in the current work area. Specify each index as a literal filename or as a character expression enclosed in parentheses. The first index in the list becomes the controlling index. If you specify an <xcIndex> as an expression and the value returned is spaces or NIL, it is ignored.

ALIAS <xcAlias> specifies the name to associate with the work area when the database file is opened. You may specify the alias name as a literal name or as a character expression enclosed in parentheses. A valid <xcAlias> may be any legal identifier (i.e., it must begin with an alphabetic character and may contain numeric or alphabetic characters and the underscore). Within a single application, CA-Clipper will not accept duplicate aliases. If this clause is omitted, the alias defaults to the database filename.

EXCLUSIVE opens the database file for nonshared use in a network environment. All other users are denied access until the database file is CLOSEd.

SHARED opens the database file for shared use in a network environment. Specifying this clause overrides the current EXCLUSIVE setting.

NEW opens <xcDatabase> in the next available work area making it the current work area. If this clause is not specified, <xcDatabase> is opened in the current work area.

READONLY opens <xcDatabase> with a read-only attribute. This lets you open database files marked read-only. If you cannot open the <xcDatabase> this way, a runtime error is generated. If this clause is not specified, <xcDatabase> is opened read-write.

VIA <cDriver> specifies the replaceable database driver (RDD) with which to process the current work area. <cDriver> is name of the RDD specified as a character expression. If <cDriver> is specified as a literal value, it must be enclosed in quotes.

If the VIA clause is omitted, the DBFNTX driver is used by default. Note that if the specified driver is not linked, an unrecoverable error occurs.

In no arguments are specified, the database file open in the current work area is closed.

Description

USE opens an existing database (.dbf) file, its associated memo (.dbt) file, and optionally associated index (.ntx or .ndx) file(s) in the current or the next available work area. In CA-Clipper, there are 250

work areas with a maximum of 255 total files open in DOS 3.3 and above. Before USE opens a database file and its associated files, it closes any active files already open in the work area. When a database file is first opened, the record pointer is positioned at the first logical record in the file (record one if there is no index file specified).

In a network environment, you may open database files as EXCLUSIVE or SHARED. EXCLUSIVE precludes the USE of the database file by other users until the file is closed. SHARED allows other users to USE the database file for concurrent access. If the database file is SHARED, responsibility for data integrity falls upon the application program. In CA-Clipper, FLOCK() and RLOCK() are the two basic means of denying other users access to a particular work area or record. If a USE is specified and neither EXCLUSIVE nor SHARED are specified, the database file is opened according to the current EXCLUSIVE setting. In CA-Clipper, all USE commands should explicitly specify how the database file is to be opened, EXCLUSIVE or SHARED. The implicit open mode specified by SET EXCLUSIVE is supplied for compatibility purposes only and not recommended.

Opening a database file in a network environment requires some special handling to be successful. First, attempt to USE the database file without specifying the INDEX list. Then, test for the success of the operation using NETERR(). If NETERR() returns false (.F.), the open operation succeeded and you can SET INDEX TO the index list. A USE will fail in a network environment if another user has EXCLUSIVE USE of the database file. Refer to the Network Programming chapter in the Programming and Utilities guide for more information on opening files in a network environment.

You can open index files with USE or SET INDEX. The first index in the list of indexes defines the current ordering of records when they are accessed. This index is referred to as the controlling index. You can change the current controlling index without closing any files by using the SET ORDER command.

To close a database and its associated files in the current work area, specify USE or CLOSE with no arguments. To close database files in all work areas, use CLOSE DATABASEs. To close index files in the current work area without closing the database file, use CLOSE INDEX or SET INDEX TO with no arguments.

Refer to the Database System section of the Basic Concepts chapter in the Programming and Utilities guide for more information about the CA-Clipper database paradigm.

Notes

- **Setting the maximum open files:** Control of the number of file handles available to a CA-Clipper application is controlled by a combination of the CONFIG.SYS FILES command, and the F parameter of the CLIPPER environment variable. The F parameter specifies the maximum number of files that can be open at any one time within the current CA-Clipper program. CA-Clipper determines the number of files that can be opened using the smaller of the two parameters. For example, if the FILES command is set to 120 and the F parameter is set to 50, the maximum number of files that can be opened is 50.

  The file limit is controlled by the operating system. Under DOS versions less than 3.3, the maximum number of files that can be opened at one time is 20 files. In DOS versions 3.3 and greater, the maximum limit is 255 files.

- **Opening the same database file in more than one work area:** Although opening a database file in more than one work area is possible in a network environment, this practice is strongly discouraged. If done, each file must be opened with a different alias, otherwise a runtime error will occur.

- **Opening two database files with the same names, in different directories:** Although opening two database files with the same names, in different directories is possible, the database files MUST have unique alias names, otherwise a runtime error will occur.

Examples

- This example opens a shared database file with associated index files in a network environment. If NETERR() returns false (.F.), indicating the USE was successful, the indexes are opened:

```
USE Accounts SHARED NEW
IF !NETERR()
   SET INDEX TO AcctNames, AcctZip
ELSE
   ? "File open failed"
   BREAK
ENDIF
```

- This example opens a database file with several indexes specified as extended expressions. Note how the array of index names is created as a constant array:

```
xcDatabase = "MyDbf"
xcIndex = {"MyIndex1", "MyIndex2", "MyIndex3"}
USE (xcDatabase) INDEX (xcIndex[1]), ;
      (xcIndex[2]), (xcIndex[3])
```

Files: Library is CLIPPER.LIB.

See Also: CLOSE NETERR() SELECT SET INDEX USED() DBUSEAREA()

WAIT*
Suspend program processing until a key is pressed

Syntax

WAIT [<expPrompt>] [TO <idVar>]

Arguments

<expPrompt> is an expression of any data type displayed as a prompt. If no <expPrompt> is specified, the default prompt displayed is: "Press any key to continue..."

TO <idVar> is the variable, of any storage class, that holds the value of the key pressed as a character value. If <idVar> does not exist or is not visible, it is created as a private variable and then assigned the character value.

Description

WAIT is a console command and wait state that displays a prompt after sending a carriage return/linefeed to the screen. It then waits for the user to press a key. If the TO clause is specified, <idVar> is assigned the keystroke as a character value. If an Alt or Ctrl key is pressed, WAIT assigns CHR(0) to <idVar>. Nonalphanumeric values entered by pressing an Alt-keypad combination assign the specified character. If the character can be displayed, it is echoed to the screen. Function keys are ignored unless assigned with SET FUNCTION or SET KEY.

WAIT is a compatibility command and therefore not recommended for general usage. It is superseded by both @...GET/READ and INKEY() for getting single character input.

Notes

. **WAITing without a prompt:** To pause execution without displaying a prompt, specify WAIT "" or INKEY(0). The latter is recommended since it doesn't disturb the current screen cursor position.

Examples

. This example illustrates how to store the WAIT keystroke as an array element:

```
aVar := ARRAY(6)

WAIT "Press a key..." TO aVar[1]
? aVar[1]               // Result: key pressed in
                        // response to WAIT
? aVar[2]               // Result: NIL
? VALTYPE(aVar)         // Result: A
? VALTYPE(aVar[1])      // Result: C
```

Files: Library is CLIPPER.LIB.

See Also: @...GET ACCEPT INKEY() INPUT MENU TO

ZAP
Remove all records from the current database file

Syntax

ZAP

Description

ZAP is a database command that permanently removes all records from files open in the current work area. This includes the current database file, index files, and associated memo file. Disk space previously occupied by the ZAPped files is released to the operating system. ZAP performs the same operation as DELETE ALL followed by PACK but is almost instantaneous.

To ZAP in a network environment, the current database file must be USEd EXCLUSIVEly. Refer to the Network Programming chapter in the Programming and Utilities guide for more information.

Examples

. This example demonstrates a typical ZAP operation in a network environment:

```
USE Sales EXCLUSIVE NEW
IF !NETERR()
   SET INDEX TO Sales, Branch, Salesman
   ZAP
   CLOSE Sales
ELSE
   ? "Zap operation failed"
   BREAK
ENDIF
```

Files: Library is CLIPPER.LIB.

See Also: DELETE PACK USE

ALERT()
Display a simple modal dialog box

Syntax

ALERT(<cMessage>, [<aOptions>]) --> nChoice

Arguments

<cMessage> is the message text displayed, centered, in the alert box. If the message contains one or more semicolons, the text after the semicolons is centered on succeeding lines in the dialog box.

<aOptions> defines a list of up to 4 possible responses to the dialog box.

Returns

ALERT() returns a numeric value indicating which option was chosen. If the Esc key is pressed, the value returned is zero.

Description

The ALERT() function creates a simple modal dialog. It is useful in error handlers and other "pause" functions. The user can respond by moving a highlight bar and pressing the Return or SpaceBar keys, or by pressing the key corresponding to the first letter of the option. If <aOptions> is not supplied, a single "Ok" option is presented.

ALERT() is sensitive to the presence or absence of the CA-Clipper full-screen I/O system. If the full-screen system is not present, ALERT() uses standard I/O to display the message and options tty-style (i.e. 80-column, without word wrap, each line ended with carriage return/linefeed).

Examples

. This example demonstrates use of an alert dialog box . First, the array of options is defined, the ALERT() function gets the user's selection, and finally, the user's choice is handled with a DO CASE...ENDCASE control structure:

```
#define AL_SAVE     1
#define AL_CANCEL   2
#define AL_CONT     3

// Define an array of options
aOptions := {"Save", "Don't Save", "Continue"}

// Display the dialog box and get the user's selection
nChoice  := ALERT("File has changed...", aOptions)

// Handle the user's request
DO CASE
CASE nChoice == AL_SAVE
   ? "Save"
CASE nChoice == AL_CANCEL
   ? "Don't Save"
CASE nChoice == AL_CONT
   ? "Continue"
OTHERWISE
   ? "Escape"
ENDCASE
//
RETURN
```

Files: Library is CLIPPER.LIB.

See Also: @...PROMPT MENU TO

ANNOUNCE
Declare a module identifier

Syntax

ANNOUNCE <idModule>

Arguments

<idModule> is a module identifier name.

Description

ANNOUNCE is a declaration statement that defines a module identifier. A linker may use this identifier later to satisfy pending module REQUESTs. ANNOUNCE and REQUEST provide a mechanism for managing application modules (.prg files).

Specify ANNOUNCE statements prior to any executable statements in a program file. A source (.prg) file can only have one module identifier; all subsequent ANNOUNCE declarations produce a compiler warning and will be ignored. Module identifiers must be unique and should not duplicate the name of any procedures or user-defined functions in a source (.prg) file.

Examples

. This example illustrates the ANNOUNCE declaration:

```
ANNOUNCE CustomInit

INIT PROCEDURE MyInit
   ? "Hypothetical Industries, Inc."
   RETURN
```

The above program module, CustomInit, should be compiled with the /N option. Subsequently, the program is addressed in the source code of another program module through use of the REQUEST statement:

```
REQUEST CustomInit
```

which causes the module CustomInit to be linked into the resultant (.exe) file.

See Also: REQUEST

BEGIN SEQUENCE
Define a sequence of statements for a BREAK

Syntax

```
BEGIN SEQUENCE
   <statements>...
[BREAK [<exp>]]
   <statements>...
[RECOVER [USING <idVar>]]
   <statements>...
END [SEQUENCE]
```

Arguments

BREAK <exp> branches execution to the statement immediately following the nearest RECOVER statement if one is specified or the nearest END SEQUENCE statement. <exp> is the value returned into the <idVar> specified in the USING clause of the RECOVER statement.

RECOVER USING <idVar> defines a recover point in the SEQUENCE construct where control branches after a BREAK statement. If USING <idVar> clause is specified, <idVar> receives the value returned by the BREAK statement. In general, this is an error object.

END defines the end point of the SEQUENCE control structure. If no RECOVER statement is specified, control branches to the first statement following the END statement after a BREAK.

Description

BEGIN SEQUENCE...END is a control structure used for exception and runtime error handling. It delimits a block of statements, including invoked procedures and user-defined functions. When a BREAK is encountered anywhere in a block of statements following the BEGIN SEQUENCE statement up to the corresponding RECOVER statement, control branches to the program statement immediately following the RECOVER statement. If a RECOVER statement is not specified, control branches to the statement following the END statement, terminating the SEQUENCE. If control reaches a RECOVER statement without encountering a BREAK, it branches to the statement following the corresponding END.

The RECOVER statement optionally receives a parameter passed by a BREAK statement that is specified with a return value. This is usually an error object, generated and returned by the current error handling block defined by ERRORBLOCK(). If an error object is returned, it can be sent messages to query information about the error. With this information, a runtime error can be handled within the context of the operation rather than in the current runtime error handler. See the example below.

Within a SEQUENCE construct there are some restrictions on what statements are allowed between the BEGIN SEQUENCE and RECOVER statements. You cannot RETURN, LOOP, or EXIT between a BEGIN SEQUENCE and RECOVER statement. From within the RECOVER statement block, however, you can LOOP, EXIT, BREAK, or RETURN since the SEQUENCE is essentially completed at that point. Using LOOP from with the RECOVER statement block is useful for reexecuting the SEQUENCE statement block. See the example below.

SEQUENCE constructs are quite flexible. They can be nested and more than one can be defined in the same procedure or user-defined function. If more than one SEQUENCE construct is specified, each SEQUENCE should delimit one discrete operation.

For more information on error objects, refer to Error class in this chapter.

Examples

. This code fragment demonstrates a SEQUENCE construct in which the BREAK occurs within the current procedure:

```
BEGIN SEQUENCE
   <statements>...
   IF lBreakCond
      BREAK
   ENDIF
RECOVER
   <recovery statements>...
END

<recovery statements>...
```

. This example demonstrates an error handler returning an error object to the variable specified in the USING clause of the RECOVER statement:

```
LOCAL objLocal, bLastHandler
//
// Save current and set new error handler
bLastHandler := ERRORBLOCK({ |objErr| ;
      MyHandler(objErr, .T.) })
//
BEGIN SEQUENCE
   .
   . <operation that might fail>
   .
RECOVER USING objLocal
   //
   // Send messages to objLocal and handle the error
   ? "Error: "
   IF objLocal:genCode != 0
      ?? objLocal:description
   ENDIF
   .
   .
   .
END
//
// Restore previous error handler
ERRORBLOCK( bLastHandler )

FUNCTION MyHandler( objError, lLocalHandler )
   //
   // Handle locally returning the error object
   IF lLocalHandler
      BREAK objError
   ENDIF
   .
   . <other statements to handle the error>
   .
   RETURN NIL
```

. This example reexecutes a SEQUENCE statement block by LOOPing from within the RECOVER statement block:

```
DO WHILE .T.
   BEGIN SEQUENCE
      .
      . <operation that may fail>
      .
   RECOVER
      IF PrintRecover()
         LOOP      // Repeat the SEQUENCE statement block
      ENDIF
```

```
      END
      EXIT            // Escape from the operation
   ENDDO
```

See Also: Error ERRORBLOCK() RETURN

DECLARE*
Create and initialize private memory variables and arrays

Syntax

DECLARE <identifier> [[:= <initializer>], ...]

Arguments

<identifier> is the name of a private variable or array to create. If the <identifier> is followed by square brackets ([ ]), it is created as an array. If the <identifier> is an array, the syntax for specifying the number of elements for each dimension is either array[<nElements>, <nElements2>,...] or array[<nElements>][<nElements2>]... The maximum number of elements per dimension is 4096.

<initializer> is the optional assignment of a value to a new private variable. An <initializer> expression for a private variable consists of the inline assignment operator (:=) followed by any valid CA-Clipper expression, including a literal array. If no explicit <initializer> is specified, the variable is given an initial value of NIL. In the case of an array, each element is NIL. Array identifiers, cannot be given values with an <initializer>.

DECLARE can create, and optionally initialize, a list of variable arrays, if definitions are separated by commas.

Description

DECLARE is a compatibility statement that is a synonym for the PRIVATE statement. Its general use is not recommended. PRIVATE should be used in all instances. See PRIVATE for more information.

See Also: PRIVATE

DO*
Call a procedure

Syntax

DO <idProcedure> [WITH <argument list>]

Arguments

<idProcedure> is the name of the procedure or user-defined function to execute.

WITH <argument list> specifies up to 128 arguments, separated by commas, to pass to <idProcedure>. Each argument may be a single variable, field, array, array element, expression, or object. Arguments can be skipped or left off the end of the list.

Description

The DO statement calls a procedure or user-defined function, optionally passing arguments to the called routine. It performs the same action as a user-defined function or procedure specified on a line by itself with the exception that variables other than field variables are passed by reference as the default. In order to pass a field variable as an argument, enclose it in parentheses, unless you declare it with the FIELD statement or with an alias.

In CA-Clipper, the number of specified arguments need not match the number of specified parameters in the called procedure. If the number of arguments is less than the number of parameters, the parameter variables with no corresponding arguments are initialized with a NIL value when the procedure is called. Also, skipping an argument within the <argument list> by leaving an empty spot next to the comma initializes the corresponding argument to NIL. To detect the position of the last argument passed in the <argument list>, use PCOUNT(). To detect a skipped argument, compare the receiving parameter to NIL.

In addition to calling a procedure or user-defined function, DO also has an effect on compilation if you compile the current program file without the /M option. If the CA-Clipper compiler encounters a DO statement and the specified procedure has not already been compiled, the compiler searches the current directory for a (.prg) file with the same name and compiles it. If the file with the same name as the procedure is not found, the called procedure is assumed to be external, and a reference is added to the object (.OBJ) file. At link time, the linker will search other object files and libraries for this external reference.

In CA-Clipper, DO is a compatibility statement and therefore not recommended. Calling a procedure or function on a line by itself is the preferred method. Since this preferred calling convention normally passes parameters by value, you must preface an argument with the pass-by-reference operator (@) in order to pass by reference. If you are using DO to make a procedure call more readable, a user-defined command, specified with the #command directive, can provide greater readability without sacrificing the safety of variables passed as parameters.

For more information on passing parameters refer to the Functions and Procedures section of the Basic Concepts chapter in the Programming and Utilities guide.

Examples

. This example executes a procedure with no parameters:

```
DO AcctsRpt
AcctsRpt()                          // Preferred method
```

. This example executes a procedure passing two constants:

```
DO QtrRpt WITH "2nd", "Sales Division"
QtrRpt("2nd", "Sales Division")     // Preferred method
```

. In this example, a procedure is executed with the first argument passed by value and the second passed by reference:

```
nNumber := 12
DO YearRpt WITH nNumber + 12, nNumber
YearRpt(nNumber + 12, @nNumber)     // Preferred method
```

. Here, a procedure is invoked with skipped arguments embedded in the list of arguments:

```
DO DisplayWindow WITH ,,,,"My Window"
DisplayWindow(,,,,"My Window")      // Preferred method
```

See Also: FUNCTION LOCAL PARAMETERS PROCEDURE PUBLIC RETURN

DO CASE
Execute one of several alternative blocks of statements

Syntax

```
DO CASE
CASE <lCondition1>
   <statements>...
[CASE <lCondition2>]
   <statements>...
[OTHERWISE]
   <statements>...
END[CASE]
```

Arguments

CASE <lCondition> defines a block of statements to execute if <lCondition> evaluates to true (.T.).

OTHERWISE defines a block of statements to execute if none of the specified CASE conditions evaluates to true (.T.).

Description

DO CASE...ENDCASE is a control structure that executes one of several blocks of statements depending on which of the associated conditions is true (.T.). It works by branching execution to the statements following the first CASE <lCondition> that evaluates to true (.T.). Execution continues until the next CASE, OTHERWISE, or ENDCASE is encountered. Control then branches to the first statement following the next ENDCASE statement.

If none of the CASE conditions evaluates to true (.T.), the statements following the OTHERWISE statement are executed up to the matching ENDCASE statement. If an OTHERWISE statement is omitted, control branches to the first statement following the matching ENDCASE statement.

You may nest any number of statements, including other control structures (i.e., DO WHILE and FOR), within a single DO CASE structure. In addition, within a single DO CASE structure, there is no fixed limit on the number of CASE statements that a DO CASE structure may contain.

DO CASE...ENDCASE is identical to IF...ELSEIF...ENDIF with neither syntax having a specific advantage over the other.

Examples

. This example uses DO CASE in a menu structure to branch control based on user selection:

```
@ 3, 25 PROMPT "First choice"
@ 4, 25 PROMPT "Second choice"
MENU TO nChoice
//
DO CASE
CASE nChoice = 0
   RETURN
CASE nChoice = 1
   ChoiceOne()
CASE nChoice = 2
   ChoiceTwo()
ENDCASE
```

See Also: BEGIN SEQUENCE DO WHILE FOR IF IF()

DO WHILE
Execute a loop while a condition is true (.T.)

Syntax

```
[DO] WHILE <lCondition>
   <statements>...
   [EXIT]
   <statements>...
   [LOOP]
   <statements>...
END[DO]
```

Arguments

<lCondition> is the logical control expression for the DO WHILE loop.

EXIT unconditionally branches control from within a DO WHILE or FOR...NEXT structure to the statement immediately following the corresponding ENDDO or NEXT statement.

LOOP branches control to the most recently executed DO WHILE or FOR statement.

Description

DO WHILE...ENDDO is a control structure that executes a block of statements repetitively, as long as <lCondition> evaluates to true (.T.). When the condition evaluates to true (.T.), control passes into the structure and proceeds until an EXIT, LOOP, or ENDDO is encountered. ENDDO returns control to the DO WHILE statement and the process repeats itself. If an EXIT statement is encountered, control branches to the nearest ENDDO or NEXT statement. If a LOOP statement is encountered, control branches to the nearest DO WHILE or FOR statement. If the condition evaluates to false (.F.), the DO WHILE construct terminates and control passes to the statement immediately following the ENDDO.

Use EXIT to terminate a DO WHILE structure based on a condition other than the DO WHILE condition. LOOP, by contrast, prevents execution of statements within a DO WHILE based on an intermediate condition, and returns to the most recent DO WHILE statement.

DO WHILE constructs may be nested within any other control structures to any depth. The only requirement is that each control structure be properly nested.

Examples

. This example demonstrates a typical control structure for a simple grouped report:

```
LOCAL cOldSalesman, nTotalAmount
USE Sales INDEX Salesman NEW
DO WHILE .NOT. EOF()
   cOldSalesman := Sales->Salesman
   nTotalAmount := 0
   DO WHILE cOldSalesman = Sales->Salesman ;
      .AND. (.NOT. EOF())
      ? Sales->Salesman, Sales->Amount
      nTotalAmount := nTotalAmount + Sales->Amount
      SKIP
   ENDDO
   ? "Total: ", nTotalAmount, "for", cOldSalesman
ENDDO
CLOSE Sales
```

- This code fragment demonstrates how LOOP can be used to provide an intermediate processing condition:

```
DO WHILE <lCondition>
   <initial processing>...
   IF <intermediate condition>
      LOOP
   ENDIF
   <continued processing>...
ENDDO
```

- This example demonstrates the use of DO WHILE to emulate a repeat until looping construct:

```
LOCAL lMore := .T.
DO WHILE lMore
   <statements>...
   lMore := (<lCondition>)
ENDDO
```

- This example uses a DO WHILE loop to move sequentially through a database file:

```
DO WHILE .NOT. EOF()
   <statements>...
   SKIP
ENDDO
```

See Also: BEGIN SEQUENCE DBEVAL() DO CASE FOR IF IF() RETURN

EXIT PROCEDURE
Declare an exit procedure

Syntax

```
EXIT PROCEDURE <idProcedure>
   [FIELD <idField list> [IN <idAlias>]]
   [LOCAL <identifier> [[:= <initializer>]]]
   [MEMVAR <identifer list>]
   .
   . <executable statements>
   .
   [RETURN]
```

Arguments

EXIT PROCEDURE declares a procedure that will be executed on program termination.

<idProcedure> is the name of the exit procedure to declare. Exit procedure names can be any length, but only the first 10 characters are significant. Names may not begin with an underscore but can contain any combination of characters, numbers, or underscores.

FIELD declares a list of identifiers to use as field names whenever encountered. If the IN clause is specified, referring to the declared name includes an implicit reference to the specified alias.

LOCAL declares and optionally initializes a list of variables or arrays whose visibility and lifetime is the current procedure.

MEMVAR declares a list of identifiers to use as private or public memory variables or arrays whenever encountered.

RETURN passes control to the next exit procedure or to the operating system, if no other exit procedures are pending.

Description

The EXIT PROCEDURE statement declares a procedure that will be executed upon program termination. EXIT procedures are called after the last executable statement in a CA-Clipper application has completed. EXIT PROCEDUREs can be used to perform common housekeeping tasks such as saving configuration settings to a file, closing a log file or concluding a communications session.

The visibility of exit procedures is restricted to the system; therefore, it is not possible to call an EXIT PROCEDURE from a procedure or user-defined function. Exit procedures do not receive parameters.

Once the last executable statement has completed, control passes from one EXIT PROCEDURE to the next until all procedures in the exit list have been called. Control then passes to the operating system.

The ANNOUNCE statement declares a module identifier for a source file (.prg). Once declared, EXIT PROCEDUREs are referenced with this module identifier. An application may use any number of exit procedures by explicitly REQUESTing their module identifiers.

The EXIT PROCEDUREs requested for an appplication are collectively referred to as the exit list. There is no mandatory execution order of procedures in the exit list; however, if an EXIT PROCEDURE is declared in the same source file (.prg) as the application's primary routine (root), it is guaranteed to be the first exit procedure called.

Termination of a given CA-Clipper application can be attributed to any of the following:

- RETURNing from the primary (root) routine
- the QUIT command
- issuing a BREAK without an enclosing BEGIN SEQUENCE...END
- unrecoverable error

Execution of an EXIT PROCEDURE cannot be guaranteed when the system encounters an unrecoverable error. If an error is raised during an exit procedure, the system returns to DOS. Pending exit procedures are not called.

Examples

- This example illustrates construction of a simple timing mechanism using INIT and EXIT PROCEDUREs:

```
// prints the amount of time required to read,
// sort, and display a list of file names.

ANNOUNCE MySystem

STATIC nStart

PROCEDURE Main()
   AEVAL( ASORT( DIRECTORY( "*.*" ) ),;
      { | aFileInfo | QOUT( aFileInfo[ 1 ] ) } )
   RETURN

INIT PROCEDURE MyInit()
   nStart := SECONDS()
   RETURN

EXIT PROCEDURE MyExit()
   ?
   ? "Elapsed Time: "
   ?? SECONDS() - nStart
   RETURN
```

See Also: ANNOUNCE REQUEST INIT PROCEDURE

EXTERNAL*
Declare a list of procedure or user-defined function names to the linker

---

Syntax

EXTERNAL <idProcedure list>

Arguments

<idProcedure list> is the list of procedures, user-defined functions, or format procedures to add to the list of routines that will be linked into the current executable (.EXE) file.

Description

EXTERNAL is a declaration statement that specifies uncoded references to the linker. Like all other declaration statements, an EXTERNAL statement must be specified before any executable statements in either the program file, or a procedure or user-defined function definition.

During the compilation of CA-Clipper source code, all explicit references to procedures and user-defined functions are made to the linker. In some instances, there may be no references made to procedure or user-defined function names until runtime. EXTERNAL resolves this by forcing the named procedures or user-defined functions to be linked even if they are not explicitly referenced in the source file. This is important in several instances:

- Procedures, user-defined functions, or formats referenced with macro expressions or variables
- Procedures and user-defined functions used in REPORT and LABEL FORMs and not referenced in the source code
- User-defined functions used in index keys and not referenced in the source code
- ACHOICE(), DBEDIT(), or MEMOEDIT() user functions

To group common EXTERNAL declarations together, place them in a header file then #include the header file into each program (.prg) file that might indirectly use them.

EXTERNAL is a compatibility statement and therefore not recommended. It is superseded by the REQUEST statement that defines a list of module identifiers to the linker.

Examples

- These examples are equivalent header files consisting of common EXTERNAL references for REPORT FORMs:

```
// Externals.ch

EXTERNAL HARDCR
EXTERNAL TONE
EXTERNAL MEMOTRAN
EXTERNAL STRTRAN

// Externals.ch
EXTERNAL HARDCR, TONE, MEMOTRAN, STRTRAN
```

See Also: #include REQUEST

FIELD
Declare database field names

Syntax

FIELD <idField list> [IN <idAlias>]

Arguments

<idField list> is a list of names to declare as fields to the compiler.

IN <idAlias> specifies an alias to assume when there are unaliased references to the names specified in the <idField list>. Unaliased references to variables in <idField list> are treated as if they are preceded by the special field alias (FIELD->).

Description

The FIELD statement declares the names of database fields to the compiler, and optionally supplies an implicit alias for each name. This allows the compiler to resolve references to variables with no explicit alias--by implicitly assuming the specified <idAlias>. Only explicit, unaliased references to the specified fields in the <idField list> are affected. The FIELD statement, like all declarations, has no effect on references made within macro expressions or variables.

The FIELD statement neither opens a database file nor verifies the existence of the specified fields. It is useful primarily to ensure correct references to fields whose accessibility is known at runtime. Attempting to access the fields when the associated database is not in USE will cause an error.

The scope of the FIELD declaration is the procedure or function in which it occurs, or the entire program (.prg) file if the declaration precedes all PROCEDURE or FUNCTION declarations and the /N compiler option is specified.

FIELD statements like other declarations must precede any executable statements in the procedure or function definition, or the program (.prg) file if the declaration has filewide scope.

FIELD used with the /W compiler option performs compile-time checking for undeclared variables.

For more information on variable declarations and scoping, refer to the Variables section in the Basic Concepts chapter of the Programming and Utilities guide.

Examples

. This user-defined function includes statements to declare database field names in both the current and Employee work areas:

```
FUNCTION DisplayRecord
   FIELD CustNo, OrderNo, Salesman
   FIELD EmpName, EmpCode IN Employee
   USE Employee NEW
   USE Orders NEW
   //
   ? CustNo             // Refers to Orders->CustNo
   ? EmpName            // Refers to Employee->EmpName
   //
   CLOSE Orders
   CLOSE Employee
   RETURN NIL
```

See Also: FUNCTION LOCAL MEMVAR PROCEDURE STATIC

FOR
Execute a block of statements a specified number of times

Syntax

```
FOR <idCounter> := <nStart> TO <nEnd>
   [STEP <nIncrement>]
   <statements>...
   [EXIT]
   <statements>...
   [LOOP]
NEXT
```

Arguments

<idCounter> is the name of the loop control or counter variable. If the specified <idCounter> is not visible or does not exist, a private variable is created.

<nStart> is the initial value assigned to <idCounter>. If <nIncrement> is negative, <nStart> must be less than <nEnd>.

TO <nEnd> defines the final value of <idCounter>. If <nIncrement> is negative, <nStart> must be greater than <nEnd>; otherwise, <nStart> must be less than <nEnd>.

STEP <nIncrement> defines the amount <idCounter> is changed for each iteration of the loop. <nIncrement> can be either positive or negative. If the STEP clause is not specified, <idCounter> is incremented by one for each iteration of the loop.

EXIT unconditionally branches control from within a FOR...NEXT construct to the statement immediately following the nearest NEXT statement.

LOOP branches control to the most recently executed FOR or DO WHILE statement.

Description

FOR...NEXT is a control structure that executes a block of statements a specified number of times. The control structure loops from the initial value of <idCounter> to the boundary specified by <nEnd>, moving through the range of values of the control variable for an increment specified by <nIncrement>. All expressions in the FOR statement are reevaluated for each iteration of the loop. The <nStart> and <nEnd> expressions, therefore, can be changed as the control structure operates.

A FOR loop operates until <idCounter> is greater than <nEnd> or an EXIT statement is encountered. Control then branches to the statement following the corresponding NEXT statement. If a LOOP statement is encountered, control branches back to the current FOR statement.

If <nIncrement> is a negative value, <idCounter> is decremented rather than incremented. The FOR loop, however, continues until <idCounter> is less than <nEnd>. This means that <nEnd> must be less than <nStart> when the FOR loop begins.

FOR loops are useful for traversing arrays where <idCounter> is used as the array subscript. See the example below.

FOR...NEXT constructs may be nested within any other control structures to any depth. The only requirement is that each control structure is properly nested.

Examples

. This example traverses an array in ascending order:

```
nLenArray := LEN(aArray)
FOR i := 1 TO nLenArray
   <statements>...
NEXT
```

. To traverse an array in descending order:

```
nLenArray := LEN(aArray)
FOR i := nLenArray TO 1 STEP -1
   <statements>...
NEXT
```

See Also: AEVAL() BEGIN SEQUENCE DO CASE DO WHILE IF

FUNCTION
Declare a user-defined function name and formal parameters

Syntax

```
[STATIC] FUNCTION <idFunction>[(<idParam list>)]
   [LOCAL <identifier> [[:= <initializer>], ... ]]
   [STATIC <identifier> [[:= <initializer>], ... ]]
   [FIELD <identifier list> [IN <idAlias>]]
   [MEMVAR <identifier list>]
   .
   . <executable statements>
   .
   RETURN <exp>
```

Arguments

<idFunction> is the name of the user-defined function to declare. User-defined function names can be any length, but only the first 10 characters are significant. Names can contain any combination of characters, numbers, or underscores, but must begin with a character or an underscore. Leading underscores are not recommended since they are reserved for internal functions.

<idParam list> is the declaration of one or more parameter variables. Variables specified in this list are declared local.

STATIC FUNCTION declares a user-defined function that can be invoked only by procedures and user-defined functions declared in the same program (.prg) file.

LOCAL declares and optionally initializes a list of variables or arrays whose visibility and lifetime is the current function.

STATIC declares and optionally initializes a list of variables or arrays whose visibility is the current user-defined function and lifetime is the duration of the program.

FIELD declares a list of identifiers to use as field names whenever encountered. If the IN clause is specified, referring to the declared name includes an implicit reference to the specified alias.

MEMVAR declares a list of identifiers to use as private or public memory variables or arrays whenever encountered.

<identifier> and **<identifier list>** are labels to be used as variable or array names.

<initializer> is a value to which an array or variable is originally set in an inline expression.

RETURN <exp> passes control back to the calling procedure or user-defined function, returning the result of <exp> as the value of the function. Each function must have at least one RETURN statement that returns a value. RETURN statements can occur anywhere in the body of a function.

Description

The FUNCTION statement declares a user-defined function and an optional list of local variables to receive parameters often referred to as formal parameters. A user-defined function is a subprogram comprised of a set of declarations and statements executed whenever you refer to <idFunction> followed by an open and closed parentheses pair. A function definition begins with a FUNCTION statement which is the FUNCTION declaration and ends with the next FUNCTION statement, PROCEDURE statement, or end of file.

Functions encapsulate a computational block of code and then later create expressions using the value returned. Functions and procedures increase both readability and modularity, isolate change, and help manage complexity.

A function in CA-Clipper is the same as a procedure, except that it must return a value. The returned value can be any data type including an array, a code block, or NIL. Each function must begin with a FUNCTION statement and contain at least one RETURN statement with an argument. Function declarations cannot be nested within other function definitions. A user-defined function can be used wherever standard functions are supported, including expressions.

The visibility of function names falls into two classes. Functions that are visible anywhere in a program are referred to as public functions and declared with a FUNCTION statement. Functions that are visible only within the current program (.prg) file are referred to as static functions and declared with a STATIC FUNCTION statement. Static functions have filewide scope.

Static functions limit visibility of a function name, thereby restricting access to the function. Because of this, subsystems defined within a single program (.prg) file can provide an access protocol with a series of public functions and conceal the implementation details of the subsystem within static functions and procedures. Since the static function references are resolved at compile time, they preempt references to public functions which are resolved at link time. This assures that within a program file, a reference to a static function executes that function if there is a name conflict with a public function.

For more information on user-defined functions, variable declarations, and parameter passing, refer to the Basic Concepts chapter in the Programming and Utilities guide.

Notes

. **Calling a user-defined function:** Use the same notation to call a user-defined function as when calling a standard CA-Clipper function:

<idFunction>([<argument list>])

You can call a user-defined function within an expression or on a line by itself. If called on a line by itself, the return value is ignored.

You can also call a user-defined function as an aliased expression by prefacing it with an alias and enclosing it in parentheses:

<idAlias>->(<idFunction>(<argument list>))

When you call a user-defined function as an aliased expression, the work area associated with <idAlias> is selected, the expression is executed, and the original work area is then reselected. You can specify an aliased expression on a line by itself, as you would any other expression.

A user-defined function in CA-Clipper may call itself recursively. This means you can make a reference to a user-defined function in statements or expressions of the same user-defined function definition.

. **Parameters:** User-defined functions, like procedures, can receive parameters passed from a calling procedure, user-defined function, or DOS command line. A parameter is a place holder for a value or reference. In CA-Clipper, there are two ways to express

parameters: you can declare a list of local variable names as a part of the FUNCTION declaration (referred to as formal parameters), or you can specify a list of private variables in a separate PARAMETERS statement. Note that you cannot mix a declaration of formal parameters with a PARAMETERS statement. Attempting this will result in a fatal compiler error.

Functions receive parameters in the order passed. In CA-Clipper, the number of parameters does not have to match the number of arguments passed. You can skip arguments or omit them from the end of the argument list. A parameter not receiving a value or reference is initialized to NIL. You can skip a parameter by passing NIL. If arguments are specified, PCOUNT() returns the position of the last argument passed.

Parameters specified in a user-defined function can receive arguments passed by value or reference. The default method for expressions and variables is by value. This includes variables that contain references to arrays and objects. All variables except field variables, when prefaced with the pass-by-reference operator (@), are passed by reference. Field variables cannot be passed by reference and are always passed by value.

Examples

. This example demonstrates a user-defined function that formats numeric values as currency:

```
? Currency( 1000 )               // Result: $1,000.00

FUNCTION Currency( nNumber )
   LOCAL cNumber
   IF nNumber < 0
      cNumber := TRANSFORM(-1 * nNumber, ;
            "999,999,999,999.99")
      cNumber := PADL("($" + LTRIM(cNumber) + ")", ;
            LEN(cNumber))
   ELSE
      cNumber := TRANSFORM(nNumber, ;
            "999,999,999,999.99")
      cNumber := PADL("$" + LTRIM(cNumber), ;
            LEN(cNumber))
   ENDIF
   RETURN cNumber
```

. This example demonstrates a user-defined function that takes a character string formatted as a comma-separated list and returns an array with one element per item:

```
aList := ListAsArray("One, Two")
// Result: {"One", "Two"}

FUNCTION ListAsArray( cList )
   LOCAL nPos
   // Define an empty array
   LOCAL aList := {}
   //
   DO WHILE (nPos := AT(",", cList)) != 0
      // Add a new element
      AADD(aList, SUBSTR(cList, 1, nPos - 1))
      cList := SUBSTR(cList, nPos + 1)
   ENDDO
   AADD(aList, cList)
   //
   // Return the array
   RETURN aList
```

. This example checks for a skipped argument by comparing the

parameter to NIL:

```
FUNCTION MyFunc( param1, param2, param3 )
   IF param2 == NIL
      param2 := "default value"
   ENDIF
   .
   . <statements>
   .
   RETURN NIL
```

. This example uses the user-defined function, Currency() (defined above), as an aliased expression:

```
USE Invoices NEW
USE Customer NEW
? Invoices->(Currency(Amount))
```

See Also: LOCAL PCOUNT() PRIVATE PROCEDURE RETURN STATIC

ALIAS()
Return a specified work area alias

Syntax

ALIAS([<nWorkArea>]) --> cAlias

Arguments

<nWorkArea> is any work area number.

Returns

ALIAS() returns the alias of the specified work area as a character string. If <nWorkArea> is not specified, the alias of the current work area is returned. If there is no database file in USE for the specified work area, ALIAS() returns a null string ("").

Description

ALIAS() is a database function that determines the alias of a specified work area. An alias is the name assigned to a work area when a database file is USEd. The actual name assigned is either the name of the database file, or a name explicitly assigned with the ALIAS clause of the USE command.

ALIAS() is the inverse of the SELECT() function. ALIAS() returns the alias name given the work area number, and SELECT() returns the work area number given the alias name.

. This example returns the name of the previously selected work area:

```
USE File1 NEW ALIAS Test1
nOldArea := SELECT()
USE File2 NEW ALIAS Test2
? ALIAS( nOldArea )            // Returns Test1
```

Files: Library is CLIPPER.LIB.

See Also: SELECT SELECT() USE

IF
Execute one of several alternative blocks of statements

Syntax

```
IF <lCondition1>
   <statements>...
[ELSEIF <lCondition2>]
   <statements>...
[ELSE]
   <statements>...
END[IF]
```

Arguments

<lCondition> is a logical control expression. If it evaluates to true (.T.), all following statements are executed until an ELSEIF, ELSE, or ENDIF is encountered.

ELSEIF <lCondition> identifies statements to execute if the associated condition evaluates to true (.T.) and all preceding IF or ELSEIF conditions evaluate to false (.F.). Any number of ELSEIF statements can be specified within the same IF...ENDIF control structure.

ELSE identifies statements to execute if the IF and all preceding ELSEIF conditions evaluate to false (.F.).

Description

The IF control structure works by branching execution to statements following the first true (.T.) evaluation of the IF or any ELSEIF condition. Execution then continues until the next ELSEIF, ELSE, or ENDIF is encountered whereupon execution branches to the first statement following the ENDIF.

If no condition evaluates to true (.T.), control passes to the first statement following the ELSE statement. If an ELSE statement is not specified, control branches to the first statement following the ENDIF statement.

IF...ENDIF structures may be nested within IF...ENDIF structures and other control structure commands. These structures, however, must be nested properly.

The IF...ELSEIF...ENDIF form of the IF construct is identical to DO CASE...ENDCASE. There is no specific advantage of one syntax over the other. The IF construct is also similar to the IF() function which can be used within expressions.

Examples

. This example evaluates a number of conditions using an IF...ELSEIF...ENDIF construct:

```
LOCAL nNumber := 0
//
IF nNumber < 50
   ? "Less than 50"
ELSEIF nNumber = 50
   ? "Is equal to 50"
ELSE
   ? "Greater than 50"
ENDIF
```

See Also: BEGIN SEQUENCE DO CASE DO WHILE FOR IF()

INIT PROCEDURE
Declare an initialization procedure

Syntax

```
INIT PROCEDURE <idProcedure> [(<idParam list>)]
   [FIELD <idField list> [IN <idAlias>]]
   [LOCAL <identifier> [[:= <initializer>]]]
   [MEMVAR <identifer list>]
   .
   . <executable statements>
   .
   [RETURN]
```

Arguments

INIT PROCEDURE declares a procedure that will be executed at program startup.

<idProcedure> is the name of the initialization procedure to declare. Initialization procedure names can be any length, but only the first 10 characters are significant. Names may not begin with an underscore but can contain any combination of characters, numbers, or underscores.

<idParam list> is the declaration of one or more parameter variables. Variables specified in this list are declared local.

FIELD declares a list of identifiers to use as field names whenever encountered. If the IN clause is specified, referring to the declared name includes an implicit reference to the specified alias.

LOCAL declares, and optionally initializes, a list of variables or arrays whose visibility and lifetime is the current procedure.

MEMVAR declares a list of identifiers to use as private or public memory variables or arrays whenever encountered.

RETURN passes control to the next initialization procedure or the first executable routine in the program, if no other initialization procedures are pending.

Description

The INIT PROCEDURE statement declares a procedure that will be executed at program startup. INIT procedures are called prior to the first executable statement in a CA-Clipper application, and are useful for performing common initialization tasks such as reading configuration settings, or opening a communications port.

INIT PROCEDUREs are executed implicitly by CA-Clipper at program startup. The visibility of initialization procedures is restricted to the system; therefore, it is not possible to call an INIT PROCEDURE from a procedure or user-defined function. Each INIT PROCEDURE receives a copy of the DOS command line arguments used to invoke the application.

Control passes from one INIT PROCEDURE to the next until all procedures in the initialization list have been called. Control then passes to the first executable statement in the program.

The ANNOUNCE statement declares a module identifier for a source (.prg) file. Once declared, INIT PROCEDUREs are referenced by this module identifier. An application may use any number of initialization procedures by explicitly REQUESTing their module identifiers.

The INIT PROCEDUREs requested for an appplication are collectively referred to as the initialization list. There is no default execution

order of procedures in the initialization list; however, the following rules apply:

- The CA-Clipper initialization procedure, CLIPINIT, is always called first.

- If an INIT PROCEDURE is declared in the same source (.prg) file as the application's primary (root) routine, it will be the last initialization procedure called.

CLIPINIT is called first to establish system integrity by installing the default error recovery system (ErrorSys). Once CLIPINIT has finished executing, control passes to the next INIT PROCEDURE in the initialization list.

If an error is raised during system initialization, the system returns to DOS, and pending initialization procedures are not called.

Examples

- This example uses both INIT and EXIT PROCEDUREs to save and restore the context of the operating system. By simply REQUESTing "SaveDos" in your main source file, the context of the operating system is preserved:

```
ANNOUNCE SaveDos

#define DOS_SCREEN   1
#define DOS_ROW      2
#define DOS_COL      3
#define DOS_CURSOR   4
#define DOS_COUNT    4

STATIC saSaveDos[ SD_COUNT ]

INIT PROCEDURE dosSave()
   SAVE SCREEN TO saSaveDos[ DOS_SCREEN ]
   saSaveDos[ DOS_ROW ]    := ROW()
   saSaveDos[ DOS_COL ]    := COL()
   saSaveDos[ DOS_CURSOR ] := SETCURSOR()
   RETURN

EXIT PROCEDURE dosRestore()
   RESTORE SCREEN FROM saSaveDos[ DOS_SCREEN ]
   SETPOS   ( saSaveDos[ DOS_ROW ], saSaveDos[ DOS_COL ] )
   SETCURSOR( saSaveDos[ DOS_CURSOR ] )
   RETURN
```

See Also: ANNOUNCE REQUEST EXIT PROCEDURE

LOCAL
Declare and initialize local variables and arrays

Syntax

LOCAL <identifier> [[:= <initializer>], ...]

Arguments

<identifier> is the name of a variable or array to declare local. If the <identifier> is followed by square brackets ([ ]), it is created as an array. If the <identifier> is an array, the syntax for specifying the number of elements for each dimension can be array[<nElements>, <nElements2>,...] or array[<nElements>][<nElements2>]... The maximum number of elements per dimension is 4096. The maximum number of dimensions per array is limited only by available memory.

<initializer> is the optional assignment of a value to a new local variable. Array identifiers, however, cannot be given values with an <initializer>. An <initializer> for a local variable consists of the inline assignment operator (:=) followed by any valid CA-Clipper expression including a literal array. If no explicit <initializer> is specified, the variable is given an initial value of NIL. In the case of an array, each element is NIL.

Note: The macro operator (&) cannot be used in a LOCAL declaration statement.

Description

LOCAL is a declaration statement that declares one or more variables or arrays local to the current procedure or user-defined function and must occur before any executable statement including PRIVATE, PUBLIC, and PARAMETERS. Local variable declarations hide all inherited private variables and visible public variables with the same name. A LOCAL statement, however, that declares a variable name which is already declared causes a fatal compiler error and no object file (.OBJ) is generated. This error can happen as a result of two declarations for the same variable name in the same routine, or as the result of redeclaring a variable with filewide scope. Declaration statements include FIELD, MEMVAR, and STATIC.

Local variables are visible only within the current procedure or user-defined function, and unlike private variables, are not visible within invoked routines. Local variables are created automatically each time the procedure in which they were declared begins executing. They continue to exist and retain their values until the declaring procedure or user-defined function returns control to the code that invoked it. If a procedure or user-defined function is invoked recursively (calls itself), each recursive activation creates a new set of local variables.

The initial value of local variables and array elements is NIL if not explicitly initialized, either in the initializer list or by assignment. The initializer expression can be any valid CA-Clipper expression, including function calls. Note that an array declaration cannot have an initializer.

The maximum number of local variables in a program is limited only by available memory. Arrays, however, assigned to a local variable are still limited to 4096 elements per dimension.

For more information on variable declarations and scoping, refer to the Variables section in the Basic Concepts chapter of the Programming and Utilities guide.

Notes

- **Inspecting local variables within the Debugger:** To access local variable names within The CA-Clipper Debugger, you must compile program (.prg) files using the /B option so that local variable information is included in the object file.

- **Local parameters:** Declare a list of local parameters as a part of a FUNCTION or PROCEDURE declaration by enclosing the list of parameters in parentheses following the <idFunction>:

  ```
  FUNCTION <idFunction>(<idParam list>)
  ```

  Declaration of local parameters supersedes creation of private parameters with the PARAMETERS statement.

- **Macro expressions:** You cannot refer to local variables within macro variables and expressions. If you refer to a local variable within a macro variable, a private or public variable with the same name will be referenced instead. If no such variable exists, a runtime error will be generated.

- **Memory files:** Local variables cannot be SAVED to or RESTOREd from memory (.mem) files.

- **Type of a local variable:** Since TYPE() uses the macro operator (&) to evaluate its argument, it cannot be used to determine the type of a local or static variable or an expression containing a local or static variable reference. The VALTYPE() function provides this facility. VALTYPE() evaluates its argument and returns the type of the return value.

Examples

- This example declares two local arrays and two local variables:

  ```
  LOCAL aArray1[20, 10], aArray2[20][10], var1, var2
  ```

- This example declares two local variables with initializers. The first is initialized to a date value and the second to a literal array:

  ```
  LOCAL dWhen := DATE()
  LOCAL aVegies := {"Tomato", "Chickadee", "Butterbean"}
  ```

See Also: FUNCTION PARAMETERS PRIVATE PROCEDURE PUBLIC STATIC

MEMVAR
Declare private and public variable names

Syntax

MEMVAR <idMemvar list>

Arguments

<idMemvar list> is a list of public and private variable names to declare to the compiler.

Description

MEMVAR is a declaration statement that causes the compiler to resolve references to variables specified without an explicit alias by implicitly assuming the memory variable alias (MEMVAR->). Only explicit, unaliased references to the specified variables are affected. MEMVAR, like all declaration statements, has no effect on references made within macro expressions or variables.

The MEMVAR statement neither creates the variables nor verifies their existence. Its primary effect is to ensure correct references to variables whose existence is known to be guaranteed at runtime. At runtime, the specified variables must be created using the PRIVATE, PARAMETERS or PUBLIC statements. This can occur in the procedure containing the MEMVAR declaration or in a higher-level procedure. Attempting to access the variables before they are created will cause an error.

The scope of the MEMVAR declaration is the procedure or function in which it occurs, or the entire source file if it precedes any PROCEDURE or FUNCTION statements and the /N compiler option is used. The /N option suppresses automatic definition of a procedure with the same name as the program (.prg) file.

Like other declaration statements, MEMVAR must precede any executable statements including PARAMETERS, PUBLIC, and PRIVATE statements in a procedure or function definition; or the program (.prg) file if the declaration has filewide scope.

MEMVAR can be used in conjunction with the /W compiler option--which generates warning messages for ambiguous variable references--to perform compile-time checking for undeclared variables.

For more information on variable declarations and scoping, refer to the Variables section in the Basic Concepts chapter of the Programming and Utilities guide.

Examples

. This example demonstrates the relationship between a private and field variable with the same name. The private variable is declared with the MEMVAR statement:

```
FUNCTION Example
   MEMVAR amount, address
   PRIVATE amount := 100
   USE Customer NEW
   //
   ? amount          // Refers to amount private variable
   ? Customer->Amount
                     // Refers to Amount field variable
   //
   RETURN NIL
```

See Also: FIELD LOCAL PRIVATE PUBLIC STATIC

PARAMETERS
Create private parameter variables

Syntax

PARAMETERS <idPrivate list>

Arguments

<idPrivate list> is one or more parameter variables separated by commas. The number of receiving variables does not have to match the number of arguments passed by the calling procedure or user-defined function.

Description

The PARAMETERS statement creates private variables to receive passed values or references. Receiving variables are referred to as parameters. The values or references actually passed by a procedure or user-defined function invocation are referred to as arguments.

When a PARAMETERS statement executes, all variables in the parameter list are created as private variables and all public or private variables with the same names are hidden until the current procedure or user-defined function terminates. A PARAMETERS statement is an executable statement and therefore can occur anywhere in a procedure or user-defined function, but must follow all compile-time variable declarations, such as FIELD, LOCAL, MEMVAR, and STATIC.

Parameters can also be declared as local variables if specified as a part of the PROCEDURE or FUNCTION declaration statement (see the example). Parameters specified in this way are referred to as formal parameters. Note that you cannot specify both formal parameters and a PARAMETERS statement with a procedure or user-defined function definition. Attempting to do this results in a fatal compiler error and an object file is not generated.

In CA-Clipper the number of arguments and parameters do not have to match. If you specify more arguments than parameters, the extra arguments are ignored. If you specify fewer arguments specified than parameters, the extra parameters are created with a NIL value. If you skip an argument, the corresponding parameter is initialized to NIL. The PCOUNT() function returns the position of the last argument passed in the list of arguments. This is different than the number of parameters passed since it includes skipped parameters.

For more information on passing parameters, refer to the Functions and Procedures section in the Basic Concepts chapter of the Programming and Utilities guide.

Examples

. This user-defined function receives values passed into private parameters with a PARAMETERS statement:

```
FUNCTION MyFunc
   PARAMETERS cOne, cTwo, cThree
   ? cOne, cTwo, cThree
   RETURN NIL
```

. This example is similar, but receives values passed into local variables, by declaring the parameter variables within the FUNCTION declaration:

```
FUNCTION MyFunc( cOne, cTwo, cThree )
   ? cOne, cTwo, cThree
   RETURN NIL
```

See Also: FUNCTION LOCAL PCOUNT() PRIVATE PROCEDURE STATIC

PRIVATE
Create and initialize private memory variables and arrays

Syntax

PRIVATE <identifier> [[:= <initializer>], ...]

Arguments

<identifier> is the name of a private variable or array to create. If the <identifier> is followed by square brackets ([ ]), an array is created and assigned to the <identifier>. When the <identifier> specification indicates an array, the syntax for specifying the number of elements for each dimension can be array[<nElements>, <nElements2>,...] or array[<nElements>][<nElements2>]... The maximum number of elements per dimension is 4096. The maximum number of dimensions is limited only by available memory.

<initializer> is the optional assignment of a value to a new private variable. An array cannot be given values with an <initializer>. An <initializer> for a private variable consists of the inline assignment operator (:=) followed by any valid CA-Clipper expression including a literal array. If no explicit <initializer> is specified, the variable is initialized to NIL. In the case of an array, each element is initialized to NIL.

You can create, and optionally, initialize a list of variables and arrays with one PRIVATE statement, if the definitions are separated by commas.

Description

The PRIVATE statement creates variables and arrays visible within the current and invoked procedures or user-defined functions. The class of variable is said to have dynamic scope. Private variables exist for the duration of the active procedure or until explicitly released with CLEAR ALL, CLEAR MEMORY, or RELEASE. When a private variable or array is created, existing and visible private and public variables of the same name are hidden until the current procedure or user-defined function terminates.

Attempting to specify a PRIVATE variable that conflicts with a previous FIELD, LOCAL, or STATIC declaration of the same name results in a fatal compiler error. This is true regardless of the scope of the declaration.

PRIVATE statements are executable statements and therefore must be specified within the body of a procedure or user-defined function and must follow all variable declarations: such as FIELD, LOCAL, MEMVAR, and STATIC.

In addition to the PRIVATE statement, private variables are also created in two other ways:

. Assignment to a variable that does not exist or is not visible will create a private variable

. Parameters received using the PARAMETERS statement are created as private variables with the same lifetime and visibility

No more than 2048 private and public variables and arrays can simultaneously exist in a single program.

For more information on variable declarations and scoping, refer to the Variables section in the Basic Concepts chapter of the Programming and Utilities guide.

Notes

- **Compatibility:** The ALL, LIKE, and EXCEPT clauses of the PRIVATE statement supported by other dBASE dialects are not supported by CA-Clipper.

Examples

- This example creates two PRIVATE arrays and three other PRIVATE variables:

```
PRIVATE aArray1[10], aArray2[20], var1, var2, var3
```

- This example creates a multidimensional private array using each element addressing convention:

```
PRIVATE aArray[10][10][10], aArray2[10, 10, 10]
```

- This example uses PRIVATE statements to create and initialize arrays and variables:

```
PRIVATE aArray := { 1, 2, 3, 4 }, ;
      aArray2 := ARRAY(12, 24)
PRIVATE cChar := SPACE(10), cColor := SETCOLOR()
```

See Also: FIELD LOCAL MEMVAR PARAMETERS PUBLIC STATIC

ALLTRIM()
Remove leading and trailing spaces from a character string

Syntax

ALLTRIM(<cString>) --> cTrimString

Arguments

<cString> is the character expression to trim.

Returns

ALLTRIM() returns a character string with leading and trailing spaces removed.

Description

ALLTRIM() is a character function that removes both leading and trailing spaces from a string. It is related to LTRIM() and RTRIM() which remove leading and trailing spaces, respectively. The inverse of ALLTRIM(), LTRIM(), and RTRIM() are the PADC(), PADL(), and PADR() functions which center, left-justify, or right-justify, character strings by padding them with fill characters.

Examples

. This example creates a string with both leading and trailing spaces, and then trims them with ALLTRIM():

```
cString := SPACE(10) + "string" + SPACE(10)
? LEN(cString)                    // Result: 26
? LEN(ALLTRIM(cString))           // Result: 6
```

Files: Library is EXTEND.LIB.

See Also: LTRIM() PAD() RTRIM()

PROCEDURE
Declare a procedure name and formal parameters

Syntax

```
[STATIC] PROCEDURE <idProcedure> [(<idParam list>)]
   [FIELD <idField list> [IN <idAlias>]
   [LOCAL <identifier> [[:= <initializer>], ... ]]
   [MEMVAR <identifier list>]
   [STATIC <identifier> [[:= <initializer>], ... ]]
   .
   . <executable statements>
   .
   [RETURN]
```

Arguments

<idProcedure> is the name of the procedure to declare. Procedure names can be any length, but only the first 10 characters are significant. Names can contain any combination of characters, numbers, or underscores, but leading underscores are reserved.

<idParam list> is the declaration of one or more parameter variables. Variables specified in this list are declared local.

STATIC PROCEDURE declares a procedure that can be called only by procedures and user-defined functions declared in the same program (.prg) file.

FIELD declares a list of identifiers, <idField list>, to use as field names whenever encountered. If the IN clause is specified, referring to the declared name, <idAlias>, is a reference to the appropriate work area of the specified database.

LOCAL declares and optionally initializes a list of variables or arrays whose visibility and lifetime is the current procedure.

<identifier>, <identifier list> is a label or labels used as variable or array names. If the <identifier> is followed by square brackets ([ ]), it is created as an array. If the <identifier> is an array, the syntax for specifying the number of elements for each dimension can be array[<nElements>, <nElements2>,...] or array[<nElements>][<nElements2>]... The maximum number of elements per dimension is 4096. The maximum number of dimensions per array is limited only by available memory.

<initializer> is the value to which an optional inline assignment sets the <identifier> variable, essentially, the assignment operator, (:=), followed by any valid CA-Clipper expression, including a literal array. If no <initializer> is specified, variables are initialized to NIL. In the case of arrays, all element are initialized to NIL.

MEMVAR declares a list of identifiers, <identifier list>, to use as private or public memory variables or arrays whenever encountered.

STATIC declares and, optionally, initializes a list of variables or arrays whose visibility is the current procedure and whose lifetime is the duration of the program.

RETURN passes control back to the calling procedure or user-defined function. If a RETURN is not specified, control passes back to the calling routine when the procedure definitions ends. In all cases, the compiler terminates the procedure definition when it encounters another PROCEDURE statement, FUNCTION statement, or end of file character.

Description

The PROCEDURE statement declares a procedure and an optional list of local variables to receive parameters passed from a calling routine. A procedure is a subprogram comprised of a set of declarations and statements executed whenever you refer to <idProcedure> followed by an open and close parentheses pair or with the DO statement. A procedure definition begins with a PROCEDURE statement and ends with the next PROCEDURE statement, FUNCTION statement, or end of file.

Procedures that encapsulate computational blocks of code to provide readability and modularity, isolate change, and help manage complexity.

A procedure in CA-Clipper is the same as a user-defined function, with the exception that it always returns NIL. Each procedure must begin with a PROCEDURE statement and may optionally contain a RETURN statement to return control to the calling procedure or user-defined function. A RETURN statement, however, is not required. Procedure declarations cannot be nested within other procedure definitions.

The visibility of procedure names falls into two classes. Procedures that are visible anywhere in a program are referred to as public procedures and declared with a PROCEDURE statement. Procedures that are visible only within the current program (.prg) file are referred to as static procedures and declared with a STATIC PROCEDURE statement. Static procedures have filewide scope.

Static procedures are quite useful for a number of reasons. First, they limit visibility of a procedure name thereby restricting access to the procedure. Because of this, subsystems defined within a single program (.prg) file can provide an access protocol with a series of public procedures and conceal the implementation details of the subsystem within static procedures and functions. Second, since the static procedure references are resolved at compile time, they preempt references to public procedures and functions which are resolved at link time. This assures that, within a program file, a reference to a static procedure executes that procedure if there is a name conflict with a public procedure or function.

For more information on procedures, variable declarations, and parameter passing, refer to the Basic Concepts chapter in the Programming and Utilities guide.

Notes

. **Calling a procedure:** There are two ways to call a procedure in CA-Clipper. First, and the preferred way, is the function-calling convention. Here you call the procedure as you would a CA-Clipper function, on a line by itself:

  <idProcedure>([<argument list>])

  The second and obsolete way is the command-calling convention using the DO...WITH command. The two methods of calling procedures differ only in the default method of passing parameters. The function-calling convention passes variables by value as a default whereas the command-calling convention passes them by reference as a default.

  A procedure can also be called as an aliased expression if it is prefaced with an alias and invoked using the function-calling convention, like this:

  <idAlias> ->(<idProcedure>(<argument list>))

  When called as an aliased expression, the work area associated with <idAlias> is selected, the procedure is executed, then the original work area is reselected. Like an expression or function, an aliased procedure can be specified on a line by itself.

A procedure in CA-Clipper may call itself recursively. This means you can call a procedure in the same procedure definition.

- **Parameters:** Procedures like user-defined functions can receive parameters passed from a calling procedure, user-defined function, or the DOS command line. A parameter is a place for a value or reference. In CA-Clipper, there are two ways to receive parameters: a list of local variable names can be declared as a part of the PROCEDURE declaration (referred to as formal parameters) or a list of private variables can be specified in a separate PARAMETERS statement. Note that you cannot mix a declaration of formal parameters with a PARAMETERS statement. Attempting this will cause a fatal compiler error.

  Procedures receive parameters in the order passed. In CA-Clipper, the number of parameters need not match the number of arguments passed. Arguments can be skipped or left off the end of the argument list. A parameter not receiving a value or reference is initialized to NIL. If arguments are specified, PCOUNT() returns the position of the last argument passed.

  Parameters specified in a procedure can receive arguments passed by value or by reference. The default method for expressions and variables depends on the calling convention. With the function-calling convention, the default passing method for expressions and variables is by value. This includes variables containing references to arrays and objects. With the command-calling convention, the default method for passing variables is by reference, except for field variables which are always passed by value. Whenever a field variable is passed, it must be specified enclosed in parentheses unless declared with the FIELD statement. Failure to do so will generate a runtime error.

Examples

- This example shows a skeleton of a typical CA-Clipper procedure that uses lexical variables:

```
PROCEDURE Skeleton( cName, cClassRoom, nBones, ;
                    nJoints )
   LOCAL nCrossBones, aOnHand := {"skull", ;
                                  "metacarpals"}
   STATIC nCounter := 0
   .
   . <executable statements>
   .
   RETURN
```

- This example determines whether an argument was skipped by comparing the parameter to NIL:

```
PROCEDURE MyProc( param1, param2, param3 )
   IF param2 != NIL
      param2 := "default value"
   ENDIF
   .
   . <statements>
   .
   RETURN
```

- This example invokes the procedure, UpdateAmount(), as an aliased expression:

```
USE Invoices NEW
USE Customer NEW
Invoices->(UpdateAmount(Amount + Amount * nInterest))
```

See Also: FUNCTION LOCAL PCOUNT() PRIVATE RETURN STATIC

PUBLIC
Create and initialize public memory variables and arrays

Syntax

PUBLIC <identifier> [[:= <initializer>], ...]

Arguments

<identifier> is the name of a public variable or array to create. If the <identifier> is followed by square brackets ([ ]), it is created as an array. If the <identifier> is an array, the syntax for specifying the number of elements for each dimension can be array[<nElements>, <nElements2>,...] or array[<nElements>][<nElements2>]... The maximum number of elements per dimension is 4096. The maximum number of dimensions per array is limited only by available memory.

<initializer> is the optional assignment of a value to a new public variable. Array identifiers, however, cannot be given values with an <initializer>. An <initializer> for a public variable consists of the inline assignment operator (:=) followed by any valid CA-Clipper expression including a literal array. Except for arrays, if no <initializer> is specified, public variables are initialized to false (.F.). This is an exception to the general rule that uninitialized variables are NIL. With arrays, however, the initial value of each element is NIL.

A list of variables and arrays can be created and optionally initialized with one PUBLIC statement if each definition is separated by a comma.

Description

The PUBLIC statement creates variables and arrays visible to all procedures and user-defined functions in a program. Public variables exist for the duration of the program or until explicitly released with CLEAR ALL, CLEAR MEMORY, or RELEASE. Declaring private, local, or static variables or arrays with the same name as existing public variables temporarily hides those public variables until the overriding variables are released or are no longer visible. An attempt to create a public variable with the same name as an existing and visible private variable is simply ignored (see Notes below for an exception).

Attempting to specify a PUBLIC variable that conflicts with a previous FIELD, LOCAL, or STATIC declaration of the same name results in a fatal compiler error. This is true regardless of the scope of the declaration.

PUBLIC statements are executable statements and therefore must be specified within the body of a procedure or user-defined function definition. They also must follow all compile-time declarations: such as FIELD, LOCAL, MEMVAR, and STATIC.

The maximum number of public and private variables and arrays that can simultaneously exist in a single program is 2048.

For more information on variable declarations and scoping, refer to the Variables section in the Basic Concepts chapter of the Programming and Utilities guide.

Notes

- **PUBLIC Clipper:** To include CA-Clipper extensions in a program and still allow the program to run under dBASE III PLUS, the special public variable, Clipper, is initialized to true (.T.) when created PUBLIC.

- **Public array name conflicts with existing private variables:**

The statement, PUBLIC x[10], will not create the public array x if there is already a private or public variable x. It will, however, destroy the contents of the existing x, replacing it with a reference to a ten element array.

Examples

- This example creates two PUBLIC arrays and one PUBLIC variable:

```
PUBLIC aArray1[10, 10], var2
PUBLIC aArray2[20][10]
```

- The following PUBLIC statements create variables and initialize them with values:

```
PUBLIC cString := SPACE(10), cColor := SETCOLOR()
PUBLIC aArray := {1, 2, 3}, aArray2 := ARRAY(12, 24)
```

See Also: LOCAL MEMVAR PARAMETERS PRIVATE STATIC

REQUEST
Declare a module request list

Syntax

REQUEST <idModule list>

Arguments

<idModule list> is the list of modules that will be linked into the current executable (.EXE) file.

Description

REQUEST is a declaration statement that defines a list of module identifiers to the linker. Like all other declaration statements, a REQUEST statement must be specified before any executable statements in either the program file, or a procedure or user-defined function definition.

During the compilation of CA-Clipper source code, all explicit references to procedures and user-defined functions are made to the linker. In some instances, within a source file, there may be no references made to procedure or user-defined function names until runtime. REQUEST resolves this situation by forcing the named procedures or user-defined functions to be linked even if they are not explicitly referenced in the source file. This is important in several instances:

- Procedures, user-defined functions, or formats referenced with macro expressions or variables
- Procedures and user-defined functions used in REPORT and LABEL FORMs and not referenced in the source code
- User-defined functions used in index keys and not referenced in the source code
- ACHOICE(), DBEDIT(), or MEMOEDIT() user functions
- Initialization procedures declared with the INIT PROCEDURE statement
- Exit procedures declared with the EXIT PROCEDURE statement

To group common REQUESTs together, place them in a header file and then #include the header file into each program file (.prg) that might indirectly use them.

Examples

- This example shows a typical header file consisting of common REQUESTs for REPORT FORMs:

```
// Request.ch

REQUEST HARDCR
REQUEST TONE
REQUEST MEMOTRAN
REQUEST STRTRAN
```

See Also: ANNOUNCE EXIT PROCEDURE EXTERNAL* INIT PROCEDURE

RETURN
Terminate a procedure, user-defined function, or program

Syntax

RETURN [<exp>]

Arguments

<exp> is an expression of any type that evaluates to the return value for user-defined functions. If a user-defined function terminates without executing a RETURN statement, the return value is NIL.

Description

RETURN terminates a procedure, user-defined function, or program by returning control to either the calling procedure or user-defined function. When RETURN executes in the highest level procedure, control passes to the operating system. All private variables created and local variables declared in the current procedure or user-defined function are released when control returns to the calling procedure.

There can be more than one RETURN in a procedure or user-defined function. A procedure or user-defined function need not, however, end with a RETURN. Since user-defined functions must return values, each must contain at least one RETURN statement with an argument.

Note: A procedure or user-defined function definition is terminated by a PROCEDURE statement, a FUNCTION statement, or end of file but not by a RETURN statement

Notes

- **Arrays:** Since array is a data type like any other data type, instances of array type are really values like character strings, and therefore, can be RETURNed from a user-defined function.

- **RETURN TO MASTER:** CA-Clipper does not support RETURN TO MASTER or any other form of RETURN specifying the level of the call to return to. You can, however, simulate these operations with BEGIN SEQUENCE...END.

Examples

- These examples illustrate the general form of the RETURN statement in a procedure and in a user-defined function:

```
PROCEDURE <idProcedure>
   //
   <statements>...
   //
   RETURN

FUNCTION <idFunction>
   //
   <statements>...
   //
   RETURN <expReturn>
```

- This example returns an array, created in a user-defined function, to a calling procedure or user-defined function:

```
FUNCTION PassArrayBack
   PRIVATE aArray[10][10]
   aArray[1][1] = "myString"
   RETURN aArray
```

See Also: BEGIN SEQUENCE FUNCTION PROCEDURE PUBLIC QUIT

ALTD()
Invoke The CA-Clipper Debugger

Syntax

ALTD([<nAction>]) --> NIL

Arguments

<nAction> defines what action ALTD() performs when invoked. The following is a complete list of <nAction> values and their actions:

ALTD() Actions

Argument	Action
None	Invokes the Debugger if it is enabled
0	Disables Alt-D
1	Enables Alt-D
Other	No action

Returns

ALTD() always returns NIL.

Description

ALTD() performs differently depending on its argument as shown in the table above. For more information on using the Debugger, refer to The CA-Clipper Debugger chapter in the Programming and Utilities guide.

Examples

. This example demonstrates a series of manifest constants that can be used as arguments for ALTD() before invoking the debugger programmatically:

```
#define ALTD_DISABLE   0
#define ALTD_ENABLE    1
//
ALTD(ALTD_ENABLE)
```

Files: Library is CLIPPER.LIB.

See Also: SET ESCAPE SETCANCEL()

STATIC
Declare and initialize static variables and arrays

Syntax

STATIC <identifier> [[:= <initializer>], ...]

Arguments

<identifier> is the name of the variable or array to declare static. If the <identifier> is followed by square brackets ([ ]), it is created as an array. If the <identifier> is an array, the syntax for specifying the number of elements for each dimension can be array[<nElements>, <nElements2>,...] or array[<nElements>] [<nElements2>]... The maximum number of elements per dimension is 4096. The maximum number of dimensions is limited only by available memory.

<initializer> is the optional assignment of a value to a new static variable. An <initializer> for a static variable consists of the inline assignment operator (:=) followed by a compile-time constant expression consisting entirely of constants and operators or a literal array. If no explicit <initializer> is specified, the variable is given an initial value of NIL. In the case of an array, each element is NIL. Array identifiers cannot be given values with an <initializer>.

Note: The macro operator (&) cannot be used in a STATIC declaration statement.

Description

The STATIC statement declares variables and arrays that have a lifetime of the entire program but are only visible within the entity that creates them. Static variables are visible only within a procedure or user-defined function if declared after a PROCEDURE or FUNCTION statement. Static variables are visible to all procedures and functions in a program file (.prg) (i.e., have filewide scope) if they are declared before the first procedure or user-defined function definition in the file. Use the /N compiler option to compile a program with filewide variable scoping.

All static variables in a program are created when the program is first invoked, and all values specified in a static <initializer> are assigned to the variable before the beginning of program execution.

Declarations of static variables within a procedure or user-defined function must occur before any executable statement including PRIVATE, PUBLIC, and PARAMETERS. If a variable of the same name is declared FIELD, LOCAL, or MEMVAR within the body of a procedure or user-defined function, a compiler error occurs and no object file (.OBJ) is generated.

The maximum number of static variables in a program is limited only by available memory.

Notes

- **Inspecting static variables within the Debugger:** To access static variable names within The CA-Clipper Debugger, you must compile program files (.prg) using the /B option so that static variable information is included in the object file (.OBJ).

- **Macro expressions:** You may not refer to static variables within macro expressions or variables. If a static variable is referred to within a macro expression or variable, a private or public variable of the same name will be accessed instead. If no such variable exists, a runtime error will be generated.

- **Memory files:** Static variables cannot be SAVED to or RESTOREd from memory files (.mem).

- **Type of a static local variable:** Since TYPE() uses the macro operator (&) to evaluate its argument, you cannot use TYPE() to determine the type of a local or static variable or an expression containing a local or static variable reference. The VALTYPE() function provides this facility by evaluating the function argument and returning the data type of its return value.

Examples

- This example declares static variables both with and without initializers:

```
STATIC aArray1[20, 10], aArray2[20][10]
STATIC cVar, cVar2
STATIC cString := "my string", var
STATIC aArray := {1, 2, 3}
```

- This example manipulates a static variable within a user-defined function. In this example, a count variable increments itself each time the function is called:

```
FUNCTION MyCounter( nNewValue )
   STATIC nCounter := 0      // Initial value assigned once
   IF nNewValue != NIL
      nCounter:= nNewValue   // New value for nCounter
   ELSE
      nCounter++             // Increment nCounter
   ENDIF
   RETURN nCounter
```

- This example demonstrates a static variable declaration that has filewide scope. In this code fragment, aArray is visible to both procedures that follow the declaration:

```
STATIC aArray := {1, 2, 3, 4}

FUNCTION One
   ? aArray[1]               // Result: 1
   RETURN NIL

FUNCTION Two
   ? aArray[3]               // Result: 3
   RETURN NIL
```

See Also: FUNCTION LOCAL PARAMETERS PRIVATE PROCEDURE PUBLIC

$
Substring comparison--binary (Relational)

Syntax

<cString1> $ <cString2>

Type

Character, memo

Operands

<cString1> is a character or memo value that is searched for in <cString2>.

<cString2> is a character or memo value within which <cString1> is sought.

Description

The $ operator is a binary relational operator that performs a case-sensitive substring search and returns true (.T.) if <cString1> is found within <cString2>.

Examples

. This example illustrates the case-sensitivity of the substring operator ($):

```
? "A" $ "ABC"            // Result: .T.
? "a" $ "ABC"            // Result: .F.
```

See Also: < <= <> = (equality) == > >=

%
Modulus--binary (Mathematical)

Syntax

<nNumber1> % <nNumber2>

Type

Numeric

Operands

<nNumber1> is the dividend of the division operation.

<nNumber2> is the divisor of the division operation.

Description

% returns a number representing the remainder of <nNumber1> divided by <nNumber2>.

Notes

. Seeking the modulus of any dividend using a zero as the divisor causes a runtime error. In versions of CA-Clipper prior to Summer '87, a modulus operation with a zero divisor returned zero.

Examples

. This example shows modulus results using different operands:

```
? 3 % 0            // Result: Runtime error
? 3 % -2           // Result:  1
? -3 % 2           // Result: -1
? -3 % 0           // Result: Runtime error
? -1 % 3           // Result: -1
? -2 % 3           // Result: -2
? 2 % -3           // Result:  2
? 1 % -3           // Result:  1
```

See Also: * ** + - / = (compound) MOD()* SET FIXED

&
Macro evaluation--unary (Special)

Syntax

&<cMacroVar>[.]
&(<cMacroExp>)

Operands

<cMacroVar> can be any character variable. The period (.) is the macro terminator and indicates the end of the macro variable and distinguishes the macro variable from adjacent text in the statement.

<cMacroExp> is a character expression enclosed in parentheses. In this instance, the expression is evaluated first, and the macro operation is performed on the resulting character value. This allows the contents of fields and array elements to be compiled and run.

Description

The macro operator in CA-Clipper is a special operator that allows runtime compilation of expressions and text substitution within strings. Whenever the macro operator (&) is encountered, the operand is submitted to a special runtime compiler (the macro compiler) that compiles expressions, but not statements or commands.

Text Substitution

Whenever a reference to a private or public macro variable, embedded in a character string is encountered:

```
cMacro := "there"
? "Hello &cMacro"            // Result: Hello there
```

the variable reference is replaced by the content of the macro variable. If you specify a macro expression (e.g., &(cMacro1 + cMacro2), and the macro variable is a local, static, field variable, or an array element, it is treated as literal text and not expanded.

Compile and Run

When a macro variable or expression specified within an expression is encountered, it is treated like an expression, with the macro symbol behaving as the compile and run operator. If the macro is specified as a macro variable:

```
cMacro := "DTOC(DATE())"
? &cMacro
```

the macro compiler compiles, then executes the content of the macro variable. The compiled code is then discarded.

If you specify an expression enclosed in parentheses and prefaced by the macro operator (&):

```
? &(INDEXKEY(0))
```

the expression is evaluated and the resulting character string is compiled and run as a macro variable.

Using the macro operator, you can compile a character string containing a code block definition:

```
bBlock := &("{ |exp| QOUT(exp) }")
```

The run portion of the operation returns the code block as a value. You

may then use the code block by invoking it with the EVAL() function. This is especially significant in activations that involve extensive looping through user-defined conditions, (operations that in earlier versions of CA-Clipper required macro expansion). In those versions, the macro expression was compiled and run for each iteration of the loop. With the combination of a macro expansion and a code block EVAL(), the compilation is performed once at compile time, and the EVAL() merely executes the code block each time through the loop. The time savings at runtime can be enormous.

```
EVAL(bBlock, DATE())
```

Notes

. **Command keywords**: You cannot use the macro operator (&) to substitute or compile command keywords. However, you can redefine command keywords by modifying the command definition in STD.CH, overriding an existing command definition with a new definition using the #command directive, or redefining a command keyword using the #translate directive. In any case, you may redefine a command keyword only at compile time, not at runtime.

. **Command arguments**: In prior versions of CA-Clipper as well as in other dialects, you could use macro variables as the arguments of commands requiring literal text values. These included all file command arguments and SET commands with toggle arguments. In these instances, you can now use an extended expression enclosed in parentheses in place of the literal argument. For example:

```
xcDatabase = "Invoices"
USE &xcDatabase.
```

can be replaced with:

```
xcDatabase = "Invoices"
USE (xcDatabase)
```

It is important to use extended expressions if you are using local and static variables. Generally, commands are preprocessed into function calls with command arguments translated into function arguments as valid CA-Clipper values. Filenames in file commands, for instance, are stringified using the smart stringify result marker and passed as arguments to the functions that actually perform the desired actions. If you specify a literal or macro value as the command argument, it is stringified. If, however, the argument is an extended expression, it is written to the result text exactly as specified. This example:

```
#command RENAME <xcOld> TO <xcNew>;
=>;
      FRENAME( <(xcOld)>, <(xcNew)> )
//
RENAME &xcOld TO &xcNew
RENAME (xcOld) TO (xcNew)
```

is written to the result text as this:

```
FRENAME( "&xcOld", "&xcNew" )
FRENAME( xcOld, xcNew )
```

when preprocessed. When the macro variables are stringified, the macro variable names are hidden in the string and not compiled. Later, at runtime, they are substituted into the string and passed as arguments to the FRENAME() function. This precludes local and static macro variables since the names of the variables are not present at runtime to be substituted. Public and private variables, however, behave as expected.

- **Lists as arguments of commands**: The macro operator (&)will not fully substitute or compile a list as an argument of most commands. In particular, these are commands where an argument list is preprocessed into an array or a code block. Instances of this are arguments of the FIELDS clause and SET INDEX. An exception is the SET COLOR command which preprocesses the list of colors into a single character string and passes it to the SETCOLOR() function.

  In any case, list arguments should always be specified as extended expressions with each list argument specified:

```
LOCAL xcIndex := { "Ntx1", "Ntx2" }
SET INDEX TO (xcIndex[1]), (xcIndex[2])
```

- **Arrays:** You can use the macro operator (&) with arrays and array elements. However, because of the increased power of CA-Clipper arrays, you may find less need to use the macro operator (&) to make variable references to arrays. You can now assign array references to variables, return array references from user-defined functions, and nest array references within other arrays. You may also create arrays by specifying literal arrays or using the ARRAY() function.

  You can, therefore, make references to arrays and array elements using both macro variables and macro expressions with the restriction that you cannot make the subscript references in a PRIVATE or PUBLIC statement. Also, you cannot specify the macro operator (&) in a declaration statement, such as a LOCAL or STATIC statement. Attempting this will generate a fatal compiler error.

  This example references array elements using macro variables:

```
cName := "aArray"
nElements := 5
cNameElement := "aArray[1]"
//
PRIVATE &cName.[nElements]    // Creates "array" with 5
                              // elements
&cNameElement. := 100         // Assigns 100 to element 1
&cName.[3] := "abc"           // Assigns "abc" to element 3
```

  You can successfully apply a macro operator (&) to an array element if the reference is made using a macro expression. A macro variable reference, however, will generate a runtime error. For example, the following lists the values of all fields of the current record:

```
USE Customer NEW
aStruc := DBSTRUCT()
//
FOR nField := 1 TO LEN(aStruc)
   ? &(aStruc[nField, 1])
NEXT
```

- **Code blocks**: You can apply the macro operator (&) to a macro variable or expression in a code block in most cases. There is a restriction when the macro variable or macro expression contains a declared variable. A runtime error occurs if you specify a complex expression (an expression that contains an operator and one or more operands) that includes the macro operator (&) within a code block.

  This has important implications on the use of local and static variables in the conditional clauses of commands, since these clauses are blockified as they are written to the result text during preprocessing. This applies to all FOR and WHILE clauses, the SET FILTER command, and the SET RELATION linking expression. The general workaround is to gather the entire expression into a single macro variable then apply the macro operator (&) to the variable.

- **Macro conditions:** When using the macro operator (&) to specify conditional clauses of database commands such as FOR or WHILE clauses, there are some restrictions based on the expression's complexity and size:

  - The maximum string size the macro compiler can process is 254 characters.

  - There is a limit to the complexity of conditions (the more complex, the fewer the number of conditions you can specify).

- **Procedures and functions**: You can reference procedure and function calls using macro variables and expressions. With DO, the macro variable reference to the procedure can be all or part of the procedure name. With a call to a function (built-in or user-defined), the macro variable reference must include the function name and all of its arguments.

  In CA-Clipper, because of the added facility code blocks, all invocations of procedures and functions using the macro operator should be converted to the evaluation of code blocks. This code fragment:

  ```
  cProc := "AcctsRpt"
  .
  .
  .
  DO &cProc
  ```

  can be replaced with:

  ```
  bProc := &( "{ || AcctsRpt() }" )
  .
  .
  .
  EVAL(bProc)
  ```

  The advantage of a code block over a macro evaluation is that the result of the compilation of a string containing a code block can be saved and therefore need only be compiled once. Macro evaluations compile each time they are referenced.

- **References into overlays:** You must declare procedures and user-defined functions that are used in macro expressions and variables but not referenced elsewhere as EXTERNAL, or the linker will not include them into the executable (.EXE) file. The only exception to this rule is a prelinked procedure or user-defined function in a (.PLL) library.

- **TEXT...ENDTEXT:** Macro variables referenced within a TEXT...ENDTEXT construct are expanded. Note that a field cannot be expanded, so you must first assign the field value to a memory variable then reference the memory variable as a macro variable within the TEXT...ENDTEXT. For example:

  ```
  USE Customer NEW
  myVar := Customer->CustName
  TEXT
  This is text with a macro &myVar
  ENDTEXT
  ```

- **Nested macros:** The processing of macro variables and expressions in CA-Clipper permits nested macro definitions. For example, after assigning a macro variable to another macro variable, the original macro variable can be expanded resulting in the expansion of the second macro variable and evaluation of its contents:

```
cOne = "&cTwo"              // expand cTwo
cTwo = "cThree"             // yielding "cThree"
cThree = "hello"
//
? &cOne                     // Result: "hello"
```

See Also: ( )

ARRAY()
Create an uninitialized array of specified length

Syntax

ARRAY(<nElements> [, <nElements>...]) --> aArray

Arguments

<nElements> is the number of elements in the specified dimension. The maximum number of elements in a dimension is 4096. Arrays in CA-Clipper can have an unlimited number of dimensions.

Returns

ARRAY() returns an array of specified dimensions.

Description

ARRAY() is an array function that returns an uninitialized array with the specified number of elements and dimensions. If more than one <nElements> argument is specified, a multidimensional array is created with the number of dimensions equal to the number of <nElements> arguments specified. Any <nElements> that is itself an array creates a nested array.

In CA-Clipper, there are several ways to create an array. You can declare an array using a declaration statement such as LOCAL or STATIC; you can create an array using a PRIVATE or PUBLIC statement; you can assign a literal array to an existing variable; or you can use the ARRAY() function. ARRAY() has the advantage that it can create arrays within expressions or code blocks.

Examples

. This example creates a one-dimensional array of five elements using the ARRAY() function, and then shows the equivalent action by assigning a literal array of NIL values:

```
aArray := ARRAY(5)
aArray := { NIL, NIL, NIL, NIL, NIL }
```

. This example shows three different statements which create the same multidimensional array:

```
aArray := ARRAY(3, 2)
aArray := { {NIL, NIL}, {NIL, NIL}, {NIL, NIL} }
aArray := { ARRAY(2), ARRAY(2), ARRAY(2) }
```

. This example creates a nested, multidimensional array:

```
aArray := ARRAY(3, {NIL,NIL})
```

Files: Library is CLIPPER.LIB.

See Also: AADD() ADEL() AFILL() AINS() LOCAL PUBLIC STATIC

( )
Function or grouping indicator (Special)

Description

Parentheses (()) in expressions group certain operations for readability or to force a particular evaluation order. Parentheses also identify a function call.

When specifying the grouping operator, the item that falls within the parentheses must be a valid expression. Subexpressions may be further grouped.

For function calls, a valid function name must precede the left parenthesis, and the function arguments, if any, must be contained within the parentheses.

Examples

. This example changes the default order of expression evaluation:

```
? 5 * (10 + 6) / 2          // Result: 40
```

. The next example shows a function call:

```
x := SQRT(100)
```

See Also: &

```
*
Multiplication--binary                          (Mathematical)
```

Syntax

<nNumber1> * <nNumber2>

Type

Numeric

Operands

<nNumber1> and **<nNumber2>** are the two numeric expressions to multiply.

Description

The multiplication operator (*) is a binary operator that returns a numeric value from its operation on <nNumber1> and <nNumber2>.

Examples

. This example shows multiplication results using different operands:

```
? 3 *  0            // Result:  0
? 3 * -2            // Result: -6
? -3 *  2           // Result: -6
? -3 *  0           // Result:  0
? -1 *  3           // Result: -3
? -2 * -3           // Result:  6
? 2 * -3            // Result: -6
? 1 * -3            // Result: -3
```

See Also: % ** + - / = (compound) SET DECIMALS SET FIXED

**
Exponentiation--binary (Mathematical)

Syntax

<nNumber1> ** <nNumber2>
<nNumber1> ^ <nNumber2>

Type

Numeric

Operands

<nNumber1> is the numeric value to raise to a power.

<nNumber2> is the power to raise <nNumber1>.

Description

The exponentiation operator (**) is a binary operator that raises <nNumber1> to the power of <nNumber2>.

Examples

. This example shows exponentiation results to 16 decimal places, using different operands:

```
SET DECIMALS TO 16
?  3 **  0            // Result: 1.000000000
?  3 **  1            // Result: 3.000000000
?  3 ** -2            // Result: 0.111111111
? -3 **  2            // Result: 9.000000000
? -3 **  0            // Result: 1.000000000
? -1 **  3            // Result: 1.000000000
? -2 **  3            // Result: 8.000000000
?  2 ** -3            // Result: 0.125000000
?  1 ** -3            // Result: 1.000000000
```

See Also: % * + - / = (compound) SET DECIMALS SET FIXED

+
Addition, unary positive, concatenation (Math, Character)

Syntax

```
<nNumber1> + <nNumber2>          (addition)
<dDate>    + <nNumber>           (addition)
<cString1> + <cString2>          (concatenation)
```

Type

Character, date, memo, numeric

Operands

<nNumber1> is a numeric value to increment by **<nNumber2>**.

<dDate> is a date value to increment by **<nNumber>** days.

<cString2> is a character string to join to the end of **<cString1>**.

Description

The (+) operator performs a number of different operations depending on the data types of the operands:

- **Unary positive sign (numeric):** A numeric expression prefaced with the plus (+) operator performs no operation on the operand except to enforce a higher level of precedence than other numeric operations except the unary minus.

- **Binary addition sign (numeric, date):** If both operands are numeric, <nNumber2> is added to <nNumber1> and the result is returned as a numeric value. If either operand is date data type and the other operand numeric data type, the <nNumber> is added as days to the <dDate> and a date value is returned.

- **Concatenation (character, memo):** If both operands are character, <cString2> (the right operand) is concatenated to <cString1> (the left operand) returning a character string.

Examples

- These examples illustrate the various forms of the + operator:

```
// Binary addition (numeric)
? 1 + 1                          // Result: 2
? 1 + 0                          // Result: 1
? 0 + 1                          // Result: 1

// Binary addition (date)
? CTOD("12/12/88") + 12          // Result: 12/24/88
? 12 + CTOD("12/12/88")          // Result: 12/24/88

// Concatenation (character)
? "Hi " + "there"                // Result: Hi there
```

See Also: % * ** - / = (compound)

Syntax

++<idVar> (prefix increment)
<idVar>++ (postfix increment)

Type

Date, numeric

Operands

<idVar> is any valid CA-Clipper identifier including a field variable. If <idVar> is a field, it must either be prefaced with an alias or declared with the FIELD statement.

The prefix increment can only be performed on an initialized value of numeric or date data type.

Description

The increment operator (++) increases the value of its operand by one. This operator adds one to the value of <idVar> and assigns the new value to <idVar>.

The ++ operator can appear before or after <idVar>. Specifying the operator before <idVar> increments and assigns the value before <idVar> is used. This, prefix notation, is the most common usage. Specifying the operator after <idVar>, postfix notation, increments and assigns the value after <idVar> is used. Stated differently, postfix notation delays the assignment portion of the operation until the rest of the expression is evaluated, and prefix notation gives the assignment precedence over all other operations in the expression.

If the reference to <idVar> is ambiguous (i.e., not declared at compile time and not explicitly qualified with an alias) <idVar> is always assumed to be MEMVAR. You can assign field variables by declaring the field variable name in a FIELD statement or referring to the field name prefaced by the FIELD-> alias or the name of the work area.

Examples

. This code uses the prefix increment operator in an assignment statement. Therefore, both variables have the same value when queried:

```
nValue := 0
nNewValue := ++nValue
? nNewValue              // Result:   1
? nValue                 // Result:   1
```

. In this example, the postfix increment operator increases the first operand of the multiplication operation by one, making its value 11; however, the assignment of this new value to the nValue variable will not take place until the entire expression is evaluated. Therefore, its value is still 10 when the multiplication operation occurs, and the result of 11 * 10 is 110. Finally, when nValue is queried again after the expression is evaluated, the postfix increment assignment is reflected in its new value, 11.

```
nValue := 10
? nValue++ * nValue      // Result: 110
? nValue                 // Result:  11
```

See Also: + -- := = (compound)

-
Subtraction, unary negative, concatenation (Math, Character)

Syntax

```
<nNumber1> - <nNumber2>          (subtraction)
<dDate1>   - <dDate2>            (subtraction)
<dDate>    - <nNumber>           (subtraction)
<cString1> - <cString2>          (concatenation)
```

Type

Character, date, memo, numeric

Operands

<nNumber1> is a numeric value to decrement by **<nNumber2>**.

<dDate1> is a date value to decrement by **<dDate2>**.

<dDate> is a date value to decrement by **<nNumber>** days.

<cString2> is a character string to join to the end of **<cString1>** after trimming the trailing blanks from <cString1>.

Description

The (-) operator performs different operations depending on the data types of the operands:

- **Unary negative sign (numeric):** A numeric expression prefaced with the minus (-) operator performs the equivalent of multiplying the operand by (-1), returning a negative numeric value.

- **Binary subtraction (numeric, date):** If both operands are numeric data type, <nNumber2> is subtracted from <nNumber1> and the result is returned as a numeric value. If both operands are date data type, <dDate2> is subtracted from <dDate1> and the result is returned as a numeric value representing the number of days between the two dates. If the left operand is date data type and the right operand numeric data type, the <nNumber> is subtracted as days from <dDate> and a date value is returned. If the order of operands is reversed, a runtime error occurs.

- **Concatenation (character, memo):** If both operands are character data type, <cString2> is joined to <cString1>, returning a character string. Trailing blanks are trimmed from <cString1> and joined to the end of the return string.

Examples

- These examples illustrate the various forms of the - operator:

```
// Binary subtraction (numeric)
? 1 - 1                          // Result: 0
? 1 - 0                          // Result: 1
? 0 - 1                          // Result: -1

// Binary subtraction (date)
? CTOD("12/12/88") - 12          // Result: 11/30/88

? 12 - CTOD("12/12/88")          // Result: Runtime error

// Concatenation (character)
? "Hi " - "There" + "*"          // Result: Hithere *
```

See Also: % * ** + / = (compound)

--
Decrement--unary (Mathematical)

Syntax

--<idVar> (prefix decrement)
<idVar>-- (postfix decrement)

Type

Date, numeric

Operands

<idVar> is any valid CA-Clipper identifier, including a field variable. If <idVar> is a field, you must reference it prefaced with an alias or declare it with the FIELD statement.

In addition, you must initialize <idVar> to a value before performing the decrement operation, and it must be either numeric or date data type.

Description

The decrement operator (--) decreases the value of its operand by one. This operator subtracts one from the value of <idVar> and assigns the new value to <idVar>.

The -- operator can appear before or after <idVar>. Specifying the operator before <idVar> decrements and assigns the value before the value is used. This is prefix notation and is the most common usage. Specifying the operator after <idVar> decrements and assigns the value after it is used. This is postfix notation. Stated differently, postfix notation delays the assignment portion of the operation until the rest of the expression is evaluated, and prefix notation gives the assignment precedence over all other operations in the expression.

If the reference to <idVar> is ambiguous (i.e., not declared at compile time and not explicitly qualified with an alias), <idVar> is always assumed to be MEMVAR. You can assign field variables by declaring the field variable name in a FIELD statement or by referring to the field name prefaced by the FIELD-> alias or by the name of the work area.

Examples

. In this example of the postfix decrement operator the assignment takes place before the original variable is decremented, thus the two values are not the same when queried:

```
nValue := 1
nNewValue := nValue--
? nNewValue                 // Result:   1
? nValue                    // Result:   0
```

. In this example, the prefix decrement operator decreases the first operand of the multiplication by one, making its value 9. Then, the assignment of this new value to the nValue variable takes place before the rest of the expression is evaluated. Thus, the new value is used to perform the multiplication operation, and the result of 9 * 9 is 81:

```
nValue := 10
? --nValue * nValue         // Result:  81
? nValue                    // Result:   9
```

See Also: ++ - := = (compound)

Syntax

<idAlias>-><idField>
<idAlias>->(<exp>)
(<nWorkArea>)-><idField>
(<nWorkArea>)->(<exp>)
FIELD-><idVar>
MEMVAR-><idVar>

Operands

<idAlias> is the name of the unselected work area to access and must refer to a work area with a database file in USE.

<nWorkArea> is the number of the unselected work area to access.

<idField> is the name of a field in the specified work area.

<exp> is an expression of any data type to be executed in the specified work area. If an expression more complicated than a single field reference is used as the second operand, the expression must be enclosed in parentheses (()).

<idVar> is any valid CA-Clipper identifier. Depending on whether you specify FIELD or MEMVAR, the identifier will be forced to a database field or memory variable (public or private) reference.

Description

When used with an <idAlias> as the first operand, the alias operator (->) accesses field information or evaluates an expression in the indicated work area. The alias operator implicitly SELECTs the <idAlias> before evaluating the <idField> or <exp> operand. When the evaluation is complete, the original work area is SELECTed again. An alias reference can be in an expression or on a line by itself:

```
? Customer->Name
Customer->(UpdateTransaction())
```

Using the alias operator lets you:

. Access information from unselected work areas within expressions

. Access environmental information from unselected work areas

. Access information from unselected work areas in modes such as REPORT and LABEL FORMs

. Write more compact code

In addition to allowing expression and field evaluation in unselected work areas, the alias operator makes an explicit reference to a field or variable using either the FIELD or the MEMVAR keyword aliases. MEMVAR forces <idVar> to refer to a memory variable name, and FIELD forces it to reference a database field. These special alias identifiers allow you to avoid ambiguity when there are conflicts between field and memory variable names. Remember that a reference to a variable identifier not prefaced with an alias defaults to a field if there are both field and memory variables with the same name. To override this, use the (/V) option when compiling.

In addition to specifying the alias as an identifier, you can access the target work area using an expression that returns the work area number

if the expression is enclosed by parentheses. This lets you use work area numbers as handles which is useful for passing references to work areas without using macros, aliases, names, etc.

Examples

. This example accesses database and work area information in an unselected work area:

```
USE Customer NEW
USE Invoices NEW
? Customer->CustName             // Result: Bill Smith
? Customer->(RECNO())            // Result: 1
? Customer->(FOUND())            // Result: .F.
? Customer->(City + ", " + State + ;
   "  " + Zip)                   // Result: ShadowVille,
                                 //         CA  90415
```

. This example uses a user-defined function (MySeek()) as an operand of the alias operator for a common operation that normally requires many more statements:

```
IF Invoices->(MySeek(CustNum))
   <process customer>...
ELSE
   <process no find>...
ENDIF
RETURN

FUNCTION MySeek( cSearch )
   SEEK cSearch
RETURN (FOUND())
```

Note: This example is just an illustration of the alias operator with a user-defined function. CA-Clipper's DBSEEK() could be used instead of MySeek()

. This example explicitly references field and memory variables with the same name:

```
USE Customer NEW
MEMVAR->CustName = "Bill Smith"      // Create a memvar
                                     // CustName
LOCATE FOR MEMVAR->CustName = FIELD->CustName
```

. This example uses an expression as a work area handle to create a work area-independent database operation:

```
cTable1 := "C:Myfile.dbf"
cTable2 := "D:Myfile.dbf"
USE (cTable1) NEW
hArea1 = SELECT()
USE (cTable2) NEW
hArea2 = SELECT()
DoStuff( hArea1, hArea2 )

FUNCTION DoStuff( hArea1, hArea2 )
   LOCAL nCount, nSave
   nSave := SELECT()
   SELECT (hArea1)
   FOR nCount := 1 TO FCOUNT()
      FIELDPUT( nCount, ( hArea2 )-> ;
         ( FIELDGET( nCount )))
   NEXT
   SELECT (nSave)
   RETURN NIL
```

See Also: FIELD FIELDNAME() FIELDPOS() FIELDGET() SELECT

.AND.
Logical AND--binary (Logical)

Syntax

<lCondition1> .AND. <lCondition2>

Type

Logical

Operands

<lCondition1> and **<lCondition2>** are logical expressions to AND.

Description

The .AND. operator is a binary logical operator that executes a logical AND operation using the following modified Boolean rules:

- Returns true (.T.) if both <lCondition1> and <lCondition2> evaluate to true (.T.).
- Returns false (.F.) if either <lCondition1> and <lCondition2> evaluate to false (.F.).

Warning! In a departure from Summer '87 and other dialect behavior, CA-Clipper shortcuts the evaluation of .AND. operands. This means that <lCondition1> and <lCondition2> are both evaluated only when <lCondition1> evaluates to true (.T.). If <lCondition1> evaluates to false (.F.), the .AND. operation is false (.F.) and <lCondition2> is therefore not evaluated.

For backward compatibility, shortcutting can be suppressed by compiling with the /Z option.

Examples

- This example shows .AND. results using different operands:

```
? .T. .AND. .T.            // Result: .T.
? .T. .AND. .F.            // Result: .F.
? .F. .AND. .T.            // Result: .F.   (shortcut)
? .F. .AND. .F.            // Result: .F.   (shortcut)
```

See Also: .NOT. .OR.

.NOT.
Logical NOT--unary (Logical)

Syntax

! <lCondition>
.NOT. <lCondition>

Type

Logical

Operands

<lCondition> is a logical expression to not.

Description

The ! (not) operator is a unary logical operator that returns the logical inverse of <lCondition>.

Examples

. This example shows .NOT. results using different operands:

```
? ! (.T.)             // Result: .F.
? ! 1 > 2             // Result: .T.
? .NOT. 1 > 2         // Result: .T.
```

See Also: .AND. .OR.

.OR.
Logical OR--binary (Logical)

Syntax

<lCondition1> .OR. <lCondition2>

Type

Logical

Operands

<lCondition1> and **<lCondition2>** are logical expressions.

Description

The .OR. operator is a binary logical operator that executes a logical OR operation using the following modified Boolean rules:

- Returns true (.T.) if either <lCondition1> or <lCondition2> evaluate to true (.T.).

- Returns false (.F.) if both <lCondition1> and <lCondition2> evaluate to false (.F.).

Warning! In a departure from the Summer '87 and other dialect's behavior, CA-Clipper shortcuts the evaluation of .OR. operands. This means that <lCondition1> and <lCondition2> are both evaluated only when <lCondition1> evaluates to false (.F.). If <lCondition1> evaluates to true (.T.), the .OR. operation is true (.T.) and <lCondition2> is therefore not evaluated.

For backward compatibility, shortcutting can be suppressed by compiling with the /Z option.

Examples

- This example shows .OR. results using different operands:

```
? .T. .OR. .T.            // Result: .T.   (shortcut)
? .T. .OR. .F.            // Result: .T.   (shortcut)
? .F. .OR. .T.            // Result: .T.
? .F. .OR. .F.            // Result: .F.
```

See Also: .AND. .NOT.

/
Division--binary (Mathematical)

Syntax

<nNumber1> / <nNumber2>

Type

Numeric

Operands

<nNumber1> is the dividend of the division operation.

<nNumber2> is the divisor of the division operation.

Description

/ returns a numeric value from the division of <nNumber1> by <nNumber2>.

Note: Dividing <nNumber1> with zero value for <nNumber2> results in a runtime error.

Examples

. This example shows division results to 16 decimal places, using different operands:

```
SET DECIMALS TO 16
?  3 /  0            // Result: Runtime error
?  3 / -2            // Result: -1.500000000
? -3 /  2            // Result: -1.500000000
? -3 /  0            // Result: Runtime error
? -1 /  3            // Result: -0.333333333
? -2 /  3            // Result: -0.666666667
?  2 / -3            // Result: -0.666666667
?  1 / -3            // Result: -0.333333333
```

See Also: % * ** + - = (compound) SET DECIMALS SET FIXED

Syntax

<object>:<message>[(<argument list>)]

Type

Object

Operands

<object> is the name of the object that is to receive the message.

<message> is the name of a method or instance variable.

<argument list> is a list of parameters that are passed to the specified message. The parentheses surrounding the argument list are optional if no parameters are supplied in the message send. By convention, however, parentheses distinguish a message send with no parameters from an access to an exported instance variable.

Description

Each class defines a set of operations that can be performed on objects of that class. Perform these operations by sending a message to the object using the send operator (:).

When a message is sent to an object, the system examines the message. If the object is of a class that defines an operation for that message, the system automatically invokes a method to perform the operation on the specified object. If the class does not define a method for the specified message, a runtime error occurs.

Executing a message send produces a return value, much like a function call. The return value varies depending on the operation performed.

Examples

. In this example, myBrowse is the name of a variable that contains a TBrowse object reference. pageUp() is a method that specifies the operation to be performed. The available operations and corresponding methods vary depending on the class of the object. The classes, Error, Get, TBColumn, and TBrowse, are documented in this chapter.

```
myBrowse:pageUp()
```

See Also: Error Get TBColumn TBrowse

:=

Inline assign--binary (Assignment)

Syntax

<idVar> := <exp>

Type

All

Operands

<idVar> is a valid variable identifier of any storage class, including a field variable. If <idVar> is not visible or does not exist, a private variable is created and assigned the result of <exp>.

<exp> is the expression whose result is assigned to <idVar>.

Description

The inline assignment operator (:=) assigns values of any data type to variables of any storage class. The operator evaluates <exp>, assigns the resulting value to <idVar>, and returns the resulting value as the return value of the operation. This latter behavior permits multiple assignments such as idVar1 := idVar2 := value. Unlike the simple assign operator (=), the inline assignment operator (:=) can be used anywhere an expression or constant is allowed, including variable declaration statements. With a variable declaration statement, assignment initializes a variable. Note, however, you cannot declare an array and initialize elements within the same program statement.

If the reference to <idVar> is ambiguous (i.e., not declared at compile time and not explicitly qualified with an alias), <idVar> is always assumed to be MEMVAR. At runtime, if no private or public variable exists with the specified name, a private variable is created. You can assign field variables with the := operator by declaring the field variable name in a FIELD statement or referring to the field name prefaced by the FIELD-> alias or the name of the work area.

Examples

. Several examples of inline assignment follow:

```
LOCAL nValue := 10
//
IF (dDate := (DATE() - 1000)) = CTOD("12/20/79")
//
? SQRT(nValue := (nValue ** 2))
//
cTransNo := cSortNo := (CustId + DTOC(DATE()))
```

. This last example performs multiple assignments using the inline assignment operator as you would with the STORE command. When := is used in this way, the assignments are executed from right to left. This feature is particularly useful when you need to store the same value to many different fields, possibly in different database files:

```
CustFile->CustId := TransFile-> ;
   TransNo := (CustId + DTOC(DATE())
```

See Also: ++ -- = (assign) = (compound) STORE*

<

Less than--binary (Relational)

Syntax

<exp1> < <exp2>

Type

Character, date, logical, memo, numeric

Operands

<exp1> and **<exp2>** are expressions of the same data type to compare.

Description

< (less than) is a binary operator that compares two values of the same data type and returns true (.T.) if <exp1> is less than <exp2>.

- **Character:** The comparison is based on the underlying ASCII code. ASCII codes for alphabetic characters are ascending (e.g., the code for "A" is 65 and the code for "Z" is 90).
- **Date:** Dates are compared according to the underlying date value.
- **Logical:** False (.F.) is less than true (.T.).
- **Memo:** Treated the same as character.
- **Numeric:** Compared based on magnitude.

Examples

- These examples illustrate how the less than operator (<) behaves with different data types:

```
// Character
? "Z" < "A"              // Result: .F.
? "ZA" < "A"             // Result: .F.
? "Z" < "AZ"             // Result: .F.

// Date
? CTOD("12/12/88") < ;
   CTOD("12/11/88")      // Result: .F.

// Logical
? .F. < .T.              // Result: .T.

// Numeric
? 2 < 1                  // Result: .F.
? 1 < 2                  // Result: .T.
```

See Also: $ > <= <> = (equality) == >=

<=
Less than or equal--binary (Relational)

Syntax

<exp1> <= <exp2>

Type

Character, date, logical, memo, numeric

Operands

<exp1> and **<exp2>** are expressions of the same data type to compare.

Description

<= (less than or equal) is a binary operator that compares two values of the same data type and returns true (.T.) if <exp1> is less than or equal to <exp2>.

- **Character:** The comparison is based on the underlying ASCII code. ASCII codes for alphabetic characters are ascending (e.g., the code for "A" is 65 and the code for "Z" is 90).
- **Date:** Dates are compared according to the underlying date value.
- **Logical**: False (.F.) is less than true (.T.).
- **Memo:** Treated the same as character.
- **Numeric:** Compared based on magnitude.

Examples

- To illustrate how the less than or equal operator (<=) behaves with different data types:

```
// Character
? "Z" <= "A"             // Result: .F.
? "ZA" <= "A"            // Result: .F.
? "Z" <= "AZ"            // Result: .F.

// Date
? CTOD("12/12/88") <= ;
   CTOD("12/11/88")      // Result: .F.

// Logical
? .F. <= .T.             // Result: .T.

// Numeric

? 2 <= 1                 // Result: .F.
? 1 <= 2                 // Result: .T.
? 1 <= 1                 // Result: .T.
```

See Also: $ < <> = (equality) == > >=

<> != #
Not equal--binary (Relational)

Syntax

<exp1> <> <exp2>
<exp1> != <exp2>
<exp1> # <exp2>

Type

Character, date, logical, memo, NIL, numeric

Operands

<exp1> and **<exp2>** are expressions of the same data type or NIL to compare for inequality.

Description

The <> (not equal) is a binary operator that compares two values of the same data type and returns true (.T.) if <exp1> is not equal to <exp2> according to the following rules:

- **Character:** The comparison is based on the underlying ASCII code and is the inverse of the equal operator (=). This means that the comparison is sensitive to the current EXACT SETting. See the examples below.
- **Date:** Dates are compared according to the underlying date value.
- **Logical:** False (.F.) is not equal to true (.T.).
- **Memo:** Treated the same as character.
- **NIL:** All values compared to NIL other than NIL return true (.T.).
- **Numeric:** Compared based on magnitude.

Examples

- These examples illustrate how the not equal operator (<>) behaves with different data types:

```
// Character
SET EXACT ON

? "123" <> "12345"         // Result: .T.
? "12345" <> "123"         // Result: .T.
? "123" <> ""              // Result: .T.
? "" <> "123"              // Result: .T.
SET EXACT OFF
? "123" <> "12345"         // Result: .T.
? "12345" <> "123"         // Result: .F.
? "123" <> ""              // Result: .F.
? "" <> "123"              // Result: .T.

// Date
? CTOD("12/12/88") <> ;
   CTOD("12/12/88")        // Result: .F.

// Logical
? .T. <> .T.               // Result: .F.
? .T. <> .F.               // Result: .T.
```

```
// NIL
? NIL <> NIL                // Result: .F.
? NIL <> 12                 // Result: .T.
? NIL <> "hello"            // Result: .T.

// Numeric
? 2 <> 1                    // Result: .T.
? 1 <> 1                    // Result: .F.
```

See Also: $ < <= = (equality) == > >=

= (assign)
Simple assign--binary (Assignment)

Syntax

<idVar> = <exp>

Type

All

Operands

<idVar> is a valid variable identifier of any storage class, including a field variable. If <idVar> is not visible or does not exist, a private variable is created and assigned the result of <exp>.

<exp> is the expression whose result is assigned to <idVar>.

Description

The simple assignment operator (=) assigns a value to a variable. It is identical in operation to the STORE command that initializes a single variable and must be specified as a program statement. The inline assignment operator (:=) is like the = operator except that you can specify it within expressions. If you specify the simple assign operator (=) within an expression, it is interpreted as the equality (=) operator.

Note: You cannot initialize a specific variable using the simple assign operator (=) in a declaration statement. Only the inline assign (:=) operator can be used for this purpose.

If the reference to <idVar> is ambiguous (i.e., not declared at compile time and not explicitly qualified with an alias) <idVar> is always assumed to be MEMVAR. At runtime, if no private or public variable exists with the specified name, a private variable is created. To assign a field variable with the = operator, you must declare the field variable name in a FIELD statement or refer to the field name prefaced by the FIELD-> alias or the name of the work area.

Examples

. These examples are valid simple assignment statements:

```
nValue = 25
nNewValue = SQRT(nValue) ** 5
nOldValue = nValue
```

. In this example, the two lines are equivalent:

```
FIELD->CustAge = 20
REPLACE CustAge WITH 20
```

See Also: ++ -- := = (compound) STORE*

ASC()
Convert a character to its ASCII value

Syntax

ASC(<cExp>) --> nCode

Arguments

<cExp> is the character expression to convert to a number.

Returns

ASC() returns an integer numeric value in the range of zero to 255, representing the ASCII value of <cExp>.

Description

ASC() is a character conversion function that returns the ASCII value of the leftmost character in a character string. ASC() is used primarily on expressions requiring numeric calculations on the ASCII value of a character. CHR() and ASC() are inverse functions.

Examples

. These examples illustrate various results of ASC():

```
? ASC("A")                    // Result: 65
? ASC("Apple")                // Result: 65
? ASC("a")                    // Result: 97
? ASC("Z") - ASC("A")         // Result: 25
? ASC("")                     // Result: 0
```

Files: Library is CLIPPER.LIB.

See Also: CHR() INKEY() STR() VAL() Inkey codes

= (compound assign)
Compound assignment--binary (Assignment)

Syntax

```
<idVar>  += <cString>        (concatenation)
<idVar>  += <nNumber>        (addition)
<idVar>  -= <cString>        (concatenation)
<idVar>  -= <nNumber>        (subtraction)
<idVar>  -= <dDate>          (subtraction)
<idVar>  *= <nNumber>        (multiplication)
<idVar>  /= <nNumber>        (division)
<idVar>  %= <nNumber>        (modulus)
<idVar>  ^= <nNumber>        (exponentiation)
```

Type

Character, date, memo, numeric

Operands

<idVar> is a variable identifier of any storage class, including a field variable. The variable must be initialized to a value before performing the operation.

If the reference to <idVar> is ambiguous (i.e., not declared at compile time and not explicitly qualified with an alias), <idVar> is assumed to be MEMVAR. You can assign field variables by declaring the field variable name in a FIELD statement or referring to the field name prefaced by the FIELD-> alias or the name of the work area.

<cString> is the character string used in the operation with <idVar>.

<nNumber> is the numeric expression used in the operation with <idVar>.

<dDate> is the date value used in the operation with <idVar>.

Description

In addition to the simple and inline assignment operators (= and :=), there are a series of compound assignment operators that perform an operation then assign a value. They have the form:

<idVar> <operator>= <exp>

Each compound assignment expression is equivalent to the assignment expression:

<idVar> := (<idVar> <operator> <exp>)

For each data type that supports an operator, the compound assignment operator performs the same operation before performing the assignment operation. The following table details all of the compound operators:

Compound Operators

Operator	Defined as	Operations
a += b	a := (a + b)	concatenation, addition
a -= b	a := (a - b)	concatenation, subtraction
a *= b	a := (a * b)	multiplication
a /= b	a := (a / b)	division
a %= b	a := (a % b)	modulus
a ^= b	a := (a ^ b)	exponentiation

Note: The exponentiation operator (**) does not have a corresponding compound assignment operator. The exponentiation compound assignment operator is ^=.

Since the compound assignment operators are based on the inline assignment operator (:=), the operation returns the result of its operation as the value of the expression. This means you can use these operators within expressions just like the inline assignment operator (:=).

Examples

. These examples use the compound addition and concatenation assignment operator:

```
// Character (concatenation)
LOCAL cString := "Hello"
? cString += " there"               // Result: "Hello there"

// Date (addition)
LOCAL dDate := CTOD("12/12/90")
dDate += 12                         // Result: 12/24/90

// Numeric (addition)
LOCAL nValue := 10
? SQRT(nValue += 15)                // Result:  5
? nValue                            // Result: 25
```

. These examples use the compound subtraction and concatenation assignment operator:

```
// Character (concatenation)
LOCAL cString := "Hello   "
? cString -= "There"                // Result: HelloThere

// Date (subtraction)
LOCAL dDate := CTOD("12/24/90")
dDate -= 12                         // Result: 12/12/90

// Numeric (subtraction)
LOCAL nValue := 10
? newValue := (nValue -= 5) ** 2    // Result: 25
? nValue                            // Result:  5
```

. This example uses the compound multiplication assignment operator:

```
LOCAL nValue := 10
? newValue := (nValue *= 5) + 100   // Result: 150
? nValue                            // Result:  50
```

. This example uses the compound division assignment operator:

```
LOCAL nValue := 10
? newValue := (nValue /= 5) + 100   // Result: 102
? nValue                            // Result:   2
```

. This example uses the compound modulus assignment operator:

```
LOCAL nValue := 10
? newValue := (nValue %= 4) * 100   // Result: 200
? nValue                            // Result:   2
```

. This example uses the compound exponentiation assignment operator:

```
LOCAL nValue := 10
? newValue := (nValue ^= 3) + 50    // Result: 1050
? nValue                            // Result: 1000
```

See Also: % * ** + ++ - -- / :=

= (equality)
Equal--binary (Relational)

Syntax

<exp1> = <exp2>

Type

Character, date, logical, memo, NIL, numeric

Operands

<exp1> and **<exp2>** are expressions of the same data type to compare.

Description

The = (equal) is a binary operator that compares two values of the same data type and returns true (.T.) if <exp1> is equal to <exp2> according to the following rules:

. **Character:** The comparison is based on the underlying ASCII code. ASCII codes for alphabetic characters are ascending (e.g., the code for "A" is 65 and the code for "Z" is 90).

  When EXACT is OFF two character strings are compared according to the following rules; assume two character strings, cLeft and cRight, where the expression to test is (cLeft = cRight):

  - If cRight is null, return true (.T.).

  - If LEN(cRight) is greater than LEN(cLeft), return false (.F.).

  - Compare all characters in cRight with cLeft. If all characters in cRight equal cLeft, return true (.T.); otherwise return false (.F.).

  With EXACT ON, two strings must match exactly, except for trailing blanks.

. **Date:** Dates are compared according to the underlying date value.

. **Logical:** True (.T.) is equal to true (.T.) and false (.F.) equal to false (.F.).

. **Memo:** Treated the same as character.

. **NIL:** True (.T.) if compared to a NIL value; false (.F.) if compared to a value of any other data type.

. **Numeric:** Compared based on magnitude.

Examples

. These examples illustrate how the equal operator (=) behaves with different data types:

```
// Character
SET EXACT ON
? "123" = "123  "        // Result: .T.
? " 123" = "123"         // Result: .F.
SET EXACT OFF
? "123" = "12345"        // Result: .F.
? "12345" = "123"        // Result: .T.
? "123" = ""             // Result: .T.
```

```
? "" = "123"              // Result: .F.

// Date
? CTOD("12/12/88") = ;
   CTOD("12/12/88")       // Result: .T.

// Logical
? .T. = .T.               // Result: .T.
? .F. = .F.               // Result: .T.

// NIL
? NIL = NIL               // Result: .T.
? NIL = 12                // Result: .F.
? NIL = CTOD("")          // Result: .F.

// Numeric
? 2 = 1                   // Result: .F.
? 1 = 1                   // Result: .T.
```

See Also: $ < <= <> == > >=

==
Exactly equal--binary (Relational)

Syntax

<exp1> == <exp2>

Type

All

Operands

<exp1> and **<exp2>** are expressions of the same data type to compare.

Description

== is a binary operator that compares two values of the same data type for exact equality depending on the data type. It returns true (.T.) if <exp1> is equal to <exp2> according to the following rules:

- **Array:** Compares for identity. If <exp1> and <exp2> are variable references to the same array, returns true (.T.); otherwise returns false (.F.).

- **Character:** Comparison is based on the underlying ASCII code. ASCII codes for alphabetic characters are ascending (e.g., the code for "A" is 65 and the code for "Z" is 90). Unlike the (=) relational operator, true (.T.) is returned if <exp1> and <exp2> are exactly equal including trailing spaces; otherwise the comparison returns false (.F.). SET EXACT has no effect.

- **Date:** Dates are compared according to the underlying date value; behaves the same as the relational equality operator (=).

- **Logical:** True (.T.) is exactly equal to true (.T.), and false (.F.) is exactly equal to false (.F.).

- **Memo:** Treated the same as character.

- **NIL:** True (.T.) if compared to a NIL value; false (.F.) if compared to a value of any other data type.

- **Numeric:** Compared based on magnitude; behaves the same as the relational equality operator (=).

- **Object:** Treated the same as array.

Examples

- These examples illustrate how the == operator behaves with different data types:

```
// Character
? "A" == "A"             // Result: .T.
? "Z" == "A"             // Result: .F.
? "A" == "A "            // Result: .F.
? "AA" == "A"            // Result: .F.

// Array or object
aOne := { 1, 2, 3 }
aTwo := { 1, 2, 3 }
aThree := aOne
? aOne == aTwo           // Result: .F.
// values within the arrays are equal, but the arrays,
// themselves, are separate and distinct
```

```
? aOne == aThree          // Result: .T.
```

See Also: $ < <= <> = (equality) > >=

>
Greater than--binary (Relational)

Syntax

<exp1> > <exp2>

Type

Character, date, logical, memo, numeric

Operands

<exp1> and **<exp2>** are expressions of any type to be compared.

Description

> (greater than) is a binary operator that compares two values of the same data type and returns true (.T.) if <exp1> is greater than <exp2>.

- **Character:** The comparison is based on the underlying ASCII code. ASCII codes for alphabetic characters are ascending (e.g., the code for "A" is 65 and the code for "Z" is 90).
- **Date:** Dates are compared according to the underlying date value.
- **Logical:** True (.T.) is greater than false (.F.).
- **Memo:** Treated the same as character.
- **Numeric:** Compared based on magnitude.

Examples

- These examples illustrate how the greater than operator (>) behaves with different data types:

```
// Character
? "Z" > "A"            // Result: .T.
? "ZA" > "A"           // Result: .T.
? "Z" > "AZ"           // Result: .T.

// Date
? CTOD("12/12/88") > ;
   CTOD("12/11/88")    // Result: .T.

// Logical
? .T. > .F.            // Result: .T.


// Numeric
? 2 > 1                // Result: .T.
? 1 > 2                // Result: .F.
```

See Also: $ < <= <> = (equality) == >=

>=
Greater than or equal--binary (Relational)

Syntax

<exp1> >= <exp2>

Type

Character, date, logical, memo, numeric

Operands

<exp1> and **<exp2>** are expressions of any data type to be compared.

Description

The >= (greater than or equal to) is a binary operator that compares two values of the same data type and returns true (.T.) if <exp1> is greater than or equal to <exp2>.

- **Character:** The comparison is based on the underlying ASCII code. ASCII codes for alphabetic characters are ascending (e.g., the code for "A" is 65 and the code for "Z" is 90).
- **Date:** Dates are compared according to the underlying date value.
- **Logical:** True (.T.) is greater than false (.F.).
- **Memo:** Treated the same as character.
- **Numeric:** Compared based on magnitude.

Examples

- These examples illustrate how the greater than or equal operator (>=) behaves with different data types:

```
// Character
? "Z" >= "A"             // Result: .T.
? "ZA" >= "A"            // Result: .T.
? "Z" >= "AZ"            // Result: .T.

// Date
? CTOD("12/12/88") >= ;
   CTOD("12/11/88")      // Result: .T.


// Logical
? .T. >= .F.             // Result: .T.

// Numeric
? 2 >= 1                 // Result: .T.
? 1 >= 2                 // Result: .F.
```

See Also: $ < <= <> = (equality) == >

Syntax

@<idVar>

Operands

<idVar> is any valid CA-Clipper variable identifier, other than a database field or an array element, to pass by reference. Database fields and array elements can only be passed by value.

Description

The pass-by-reference operator (@) passes variables by reference to functions or procedures invoked with function-calling syntax. It is a unary prefix operator whose operand may be any variable name.

Passing a variable by reference means that a reference to the value of the argument is passed instead of a copy of the value. The receiving parameter then refers to the same location in memory as the argument. If the called routine changes the value of the receiving parameter, it also changes the argument passed from the calling routine.

Passing a variable by value means that the argument is evaluated and its value is copied to the receiving parameter. Changes to a receiving parameter are local to the called routine and lost when the routine terminates. The default method of passing arguments is by value for all data types including references to arrays and objects.

Examples

. This example demonstrates the difference between passing a user-defined function argument by reference and by value:

```
LOCAL nNum := 1                    // Initial values
LOCAL aArr := { "a", "b", "c" }
//
CLS
// Print initial values
? VALTYPE(nNum), nNum
? VALTYPE(aArr), aArr[1], aArr[2], aArr[3]
//

Udf1(nNum)                         // Pass by value (default)
? VALTYPE(nNum), nNum              // Result:  N, 1
//
Udf1(aArr[1])                      // Pass by value (default)
? VALTYPE(aArr), aArr[1],;
      aArr[2], aArr[3]             // Result:  A, "a" "b" "c"
//
Udf2( aArr                          // Pass a reference to
   ? VALTYPE(aArr), aArr[1],;       // the array (default)
      aArr[2], aArr[3]              // A, "Oi" "b" "c"
//
Udf1(@nNum)                         // Force pass by reference
? VALTYPE(nNum), nNum               // Result:  N, 1000
//
Udf3(@aArr)                         // Pass array by reference
? VALTYPE(aArr)                     // Result:  N
//
RETURN NIL


FUNCTION Udf1(nParam)               // Receive as a local
? nParam                            // parameter
```

```
nParam := 1000
? nParam
RETURN NIL
//
FUNCTION Udf2( aParam )            // Receive as a local
? VALTYPE(aParam), aParam[1]       // parameter
aParam[1] := "Oi!"
? aParam[1]
RETURN NIL
//
FUNCTION Udf3(aParam)              // Receive as a local
? VALTYPE(aParam), aParam[1]       // parameter
aParam := 1000
? aParam
RETURN NIL
```

See Also: FUNCTION PROCEDURE

[ ]

Array element indicator (Special)

Syntax

```
<aArray>[<nSubscript>, ... ]
<aArray>[<nSubscript1>][<nSubscript2>] ...
```

Operands

<aArray> is an expression that returns a reference to an array. This is generally a variable identifier or instance variable.

<nSubscript> is a numeric expression that addresses an individual element in the specified array or subarray. Each subscript corresponds to a dimension of the array.

Description

The subscript operator ([]) specifies a single array element. The name of a previously declared array must precede the left bracket and the array element subscript must appear as a numeric expression within the brackets. You can make array element references using Pascal or C-style syntax.

Examples

. This example accesses each element in a two-dimensional array of known dimensions:

```
LOCAL i, j
FOR i := 1 TO 5
   FOR j := 1 TO 10
      ? aOne[i, j]
   NEXT
NEXT
```

. These examples specify an <aArray> expression:

```
LOCAL aArray := { 1, 2, 3, 4, 5 }
//
? ArrayFunc()[]
? { {1, 2}, {3, 4} }[1][2]          // Result: 2
? aArray[5]                         // Result: 5
```

. This example queries and assigns a static array encapsulated within a function definition:

```
? ArrayFunc()[1]                    // Result: 1
ArrayFunc()[1] := 10
? ArrayFunc()[1]                    // Result: 10

FUNCTION ArrayFunc
   STATIC aArray := { 1, 2 }
   RETURN aArray
```

See Also: ARRAY() LOCAL PRIVATE PUBLIC STATIC

ASCAN()
Scan an array for a value or until a block returns true (.T.)

Syntax

ASCAN(<aTarget>, <expSearch>,
[<nStart>], [<nCount>]) --> nStoppedAt

Arguments

<aTarget> is the array to scan.

<expSearch> is either a simple value to scan for, or a code block. If <expSearch> is a simple value it can be character, date, logical, or numeric type.

<nStart> is the starting element of the scan. If this argument is not specified, the default starting position is one.

<nCount> is the number of elements to scan from the starting position. If this argument is not specified, all elements from the starting element to the end of the array are scanned.

Returns

ASCAN() returns a numeric value representing the array position of the last element scanned. If <expSearch> is a simple value, ASCAN() returns the position of the first matching element, or zero if a match is not found. If <expSearch> is a code block, ASCAN() returns the position of the element where the block returned true (.T.).

Description

ASCAN() is an array function that scans an array for a specified value and operates like SEEK when searching for a simple value. The <expSearch> value is compared to the target array element beginning with the leftmost character in the target element and proceeding until there are no more characters left in <expSearch>. If there is no match, ASCAN() proceeds to the next element in the array.

Since ASCAN() uses the equal operator (=) for comparisons, it is sensitive to the status of EXACT. If EXACT is ON, the target array element must be exactly equal to the result of <expSearch> to match.

If the <expSearch> argument is a code block, ASCAN() scans the <aTarget> array executing the block for each element accessed. As each element is encountered, ASCAN() passes the element's value as an argument to the code block, and then performs an EVAL() on the block. The scanning operation stops when the code block returns true (.T.), or ASCAN() reaches the last element in the array.

Examples

. This example demonstrates scanning a three-element array using simple values and a code block as search criteria. The code block criteria shows how to perform a case-insensitive search:

```
aArray := { "Tom", "Mary", "Sue" }
? ASCAN(aArray, "Mary")             // Result: 2
? ASCAN(aArray, "mary")             // Result: 0
//
? ASCAN(aArray, { |x| UPPER(x) ;
      == "MARY" })                  // Result: 2
```

. This example demonstrates scanning for multiple instances of a search argument after a match is found:

```
LOCAL aArray := { "Tom", "Mary", "Sue",;
                  "Mary" }, nStart := 1
//
// Get last array element position
nAtEnd := LEN(aArray)
DO WHILE (nPos := ASCAN(aArray, "Mary", ;
             nStart)) > 0
   ? nPos, aArray[nPos]
   //
   // Get new starting position and test
   // boundary condition
   IF (nStart := ++nPos) > nAtEnd
      EXIT
   ENDIF
ENDDO
```

. This example scans a two dimensional array using a code block. Note that the parameter aVal in the code block is an array:

```
LOCAL aArr:={}
CLS
AADD(aArr,{"one","two"})
AADD(aArr,{"three","four"})
AADD(aArr,{"five","six"})
? ASCAN(aArr, {|aVal| aVal[2] == "four"})         // Returns 2
```

Files: Library is CLIPPER.LIB.

See Also: AEVAL() EVAL()

{ }
Literal array and code block delimiters (Special)

Syntax

{ <exp list> } (literal array)
{ |<param list>| <exp list> } (code block definition)

Operands

<exp list> is a list of expressions of any type. If the item is a literal array definition, it can contain another literal array definition.

<param list> is a list of variables to receive parameters passed to a code block from an invocation of the EVAL() function. The parameter list is comma-separated and must be enclosed by vertical bars (||). Variables specified in this list are declared local to the code block and are visible only within the code block definition.

Description

Curly braces ({}) delimit references to literal arrays or code blocks. If the reference is a literal array, you can use them to create an array in either an assignment or a variable declaration statement. If the reference is a variable declaration, the array can contain expressions of any kind as elements, unless STATIC is the declaration statement. In this case, the literal array can only contain constant values.

Examples

. This example uses literal arrays in declaration statements to create a variable and initialize it with an array reference:

```
LOCAL aPerson := { "Name", "Address", DATE() }
STATIC aNumList := { 1, 2, 3, 4, 5, 6, 7, 8 }
```

. This example creates a multiple dimension literal array:

```
aMulti := { {1, 2, 3}, {"A", "B", "C"} }
```

. This example uses curly braces to formulate a simple code block with a single parameter:

```
LOCAL bSayBlock
bSayBlock := { |x| QOUT(x) }
EVAL(bSayBlock, 10)               // Result: 10
```

See Also: EVAL() LOCAL PRIVATE PUBLIC STATIC

#command | #translate
Specify a user-defined command or translation directive

Syntax

#command   <matchPattern> => <resultPattern>
#translate <matchPattern> => <resultPattern>

Arguments

<matchPattern> is the pattern the input text should match.

<resultPattern> is the text produced if a portion of input text matches the <matchPattern>.

The => symbol between <matchPattern> and <resultPattern> is, along with #command or #translate, a literal part of the syntax that must be specified in a #command or #translate directive. The symbol consists of an equal sign followed by a greater than symbol with no intervening spaces. Do not confuse this symbol with the >= or the <= comparison operators in the CA-Clipper language.

Description

#command and #translate are translation directives that define commands and pseudofunctions. Each directive specifies a translation rule. The rule consists of two portions: a match pattern and a result pattern. The match pattern matches a command specified in the program (.prg) file and saves portions of the command text (usually command arguments) for the result pattern to use. The result pattern then defines what will be written to the result text and how it will be written using the saved portions of the matching input text.

#command and #translate are similar, but differ in the circumstance under which their match patterns match input text. A #command directive matches only if the input text is a complete statement, while #translate matches input text that is not a complete statement. #command defines a complete command and #translate defines clauses and pseudofunctions that may not form a complete statement. In general, use #command for most definitions and #translate for special cases.

#command and #translate are similar to but more powerful than the #define directive. #define, generally, defines identifiers that control conditional compilation and manifest constants for commonly used constant values such as INKEY() codes. Refer to any of the header files in the \CLIPPER5\INCLUDE directory for examples of manifest constants defined using #define.

#command and #translate directives have the same scope as the #define directive. The definition is valid only for the current program (.prg) file unless defined in STD.CH or the header specified with the /U option on the compiler command line. If defined elsewhere, the definition is valid from the line where it is specified to the end of the program file. Unlike, #define, a #translate or #command definition cannot be explicitly undefined. The #undef directive has no effect on a #command or #translate definition.

As the preprocessor encounters each source line preprocessor, it scans for definitions in the following order of precedence: #define, #translate, and #command. When there is a match, the substitution is made to the result text and the entire line is reprocessed until there are no matches for any of the three types of definitions. #command and #translate rules are processed in stack-order (i.e., last in-first out, with the most recently specified rule processed first).

In general, a command definition provides a way to specify an English language statement that is, in fact, a complicated expression or

function call, thereby improving the readability of source code. You can use a command in place of an expression or function call to impose order of keywords, required arguments, combinations of arguments that must be specified together, and mutually exclusive arguments at compile time rather than at runtime. This can be important since procedures and user-defined functions can now be called with any number of arguments, forcing any argument checking to occur at runtime. With command definitions, the preprocessor handles some of this.

All commands in CA-Clipper are defined using the #command directive and supplied in the standard header file, STD.CH, located in the \CLIPPER5\INCLUDE directory. The syntax rules of #command and #translate facilitate the processing of all CA-Clipper and dBASE-style commands into expressions and function calls. This provides CA-Clipper compatibility as well as avenues of compatibility with other dialects.

When defining a command, there are several prerequisites to properly specifying the command definition. Many preprocessor commands require more than one #command directive because mutually exclusive clauses contain a keyword or argument. For example, the @...GET command has mutually exclusive VALID and RANGE clauses and is defined with a different #command rule to implement each clause.

This also occurs when a result pattern contains different expressions, functions, or parameter structures for different clauses specified for the same command (e.g., the @...SAY command). In STD.CH, there is a #command rule for @...SAY specified with the PICTURE clause and another for @...SAY specified without the PICTURE clause. Each formulation of the command is translated into a different expression. Because directives are processed in stack order, when defining more than one rule for a command, place the most general case first, followed by the more specific ones. This assures that the proper rule will match the command specified in the program (.prg) file.

For more information and a general discussion of commands, refer to the Basic Concepts chapter in the Programming and Utilities guide.

Match pattern

The <matchPattern> portion of a translation directive is the pattern the input text must match. A match pattern is made from one or more of the following components, which the preprocessor tries to match against input text in a specific way:

. **Literal values** are actual characters that appear in the match pattern. These characters must appear in the input text, exactly as specified to activate the translation directive

. **Words** are keywords and valid identifiers that are compared according to the dBASE convention (case-insensitive, first four letters mandatory, etc.). The match pattern must start with a Word.

  #xcommand and #xtranslate can recognize keywords of more than four significant letters.

. **Match markers** are label and optional symbols delimited by angle brackets (<>) that provide a substitute (idMarker) to be used in the <resultPattern> and identify the clause for which it is a substitute. Marker names are identifiers and must, therefore, follow the CA-Clipper identifier naming conventions. In short, the name must start with an alphabetic or underscore character, which may be followed by alphanumeric or underscore characters.

  This table describes all match marker forms.

  Match Markers

  Match Marker **Name**

```
-----------------------------------------------------------------
<idMarker>               Regular match marker
<idMarker,...>           List match marker
<idMarker:word list>     Restricted match marker
<*idMarker*>             Wild match marker
<(idMarker)>             Extended Expression match marker
-----------------------------------------------------------------
```

- **Regular match marker:** Matches the next legal expression in the input text. The regular match marker, a simple label, is the most general and, therefore, the most likely match marker to use for a command argument. Because of its generality, it is used with the regular result marker, all of the stringify result markers, and the blockify result marker.

- **List match marker:** Matches a comma-separated list of legal expressions. If no input text matches the match marker, the specified marker name contains nothing. You must take care in making list specifications because extra commas will cause unpredictable and unexpected results.

  The list match marker defines command clauses that have lists as arguments. Typically these are FIELDS clauses or expression lists used by database commands. When there is a match for a list match marker, the list is usually written to the result text using either the normal or smart stringify result marker. Often, lists are written as literal arrays by enclosing the result marker in curly ({ }) braces.

- **Restricted match marker:** Matches input text to one of the words in a comma-separated list. If the input text does not match at least one of the words, the match fails and the marker name contains nothing.

  A restricted match marker is generally used with the logify result marker to write a logical value into the result text. If there is a match for the restricted match marker, the corresponding logify result marker writes true (.T.) to the result text; otherwise it writes false (.F.). This is particularly useful when defining optional clauses that consist of a command keyword with no accompanying argument. STD.CH implements the REST clause of database commands using this form.

- **Wild match marker:** Matches any input text from the current position to the end of a statement. Wild match markers generally match input that may not be a legal expression, such as #command NOTE <*x*> in STD.CH, gather the input text to the end of the statement, and write it to the result text using one of the stringify result markers.

- **Extended Expression match marker:** Matches a regular or extended expression, including a filename or path specification. It is used with the smart stringify result marker to assure that extended expressions won't get stringified, while normal, unquoted string file specifications will.

. **Optional match clauses** are portions of the match pattern enclosed in square brackets ([ ]). They specify a portion of the match pattern that may be absent from the input text. An optional clause may contain any of the components allowed within a <matchPattern>, including other optional clauses.

Optional match clauses may appear anywhere and in any order in the match pattern and still match input text. Each match clause may appear only once in the input text. There are two types of optional match clauses: one is a keyword followed by match marker, and the other is a keyword by itself. These two types of optional match clauses can match all of the traditional command clauses typical of

```
the CA-Clipper command set.

Optional match clauses are defined with a regular or list match
marker to match input text if the clause consists of an argument or a
keyword followed by an argument (see the INDEX clause of the USE
command in STD.CH).  If the optional match clause consists of a
keyword by itself, it is matched with a restricted match marker (see
the EXCLUSIVE or SHARED clause of the USE command in STD.CH).

In any match pattern, you may not specify adjacent optional match
clauses consisting solely of match markers, without generating a
compiler error.  You may repeat an optional clause any number of
times in the input text, as long as it is not adjacent to any other
optional clause.  To write a repeated match clause to the result
text, use repeating result clauses in the <resultPattern> definition.
```

See Also: #command...

Result Pattern

The <resultPattern> portion of a translation directive is the text the preprocessor will produce if a piece of input text matches the <matchPattern>. <resultPattern> is made from one or more of the following components:

. **Literal tokens** are actual characters that are written directly to the result text.

. **Words** are CA-Clipper keywords and identifiers that are written directly to the result text.

. **Result markers** refer directly to a match marker name. Input text matched by the match marker is written to the result text via the result marker.

This table lists the Result marker forms.

Result Markers

Result Marker	Name
<idMarker>	Regular result marker
#<idMarker>	Dumb stringify result marker
<"idMarker">	Normal stringify result marker
<(idMarker)>	Smart stringify result marker
<{idMarker}>	Blockify result marker
<.idMarker.>	Logify result marker

- **Regular result marker:** Writes the matched input text to the result text, or nothing if no input text is matched. Use this, the most general result marker, unless you have special requirements. You can use it with any of the match markers, but it almost always is used with the regular match marker.

- **Dumb stringify result marker:** Stringifies the matched input text and writes it to the result text. If no input text is matched, it writes a null ("") string. If the matched input text is a list matched by a list match marker, this result marker stringifies the entire list and writes it to the result text.

  This result marker writes output to result text where a string is always required. This is generally the case for commands where a command or clause argument is specified as a literal value but the result text must always be written as a string even if the argument is not specified.

- **Normal stringify result marker:** Stringifies the matched input text and writes it to the result text. If no input text is matched, it writes nothing to the result text. If the matched input text is a list matched by a list match marker, this result marker stringifies each element in the list and writes it to the result text.

  The normal stringify result marker is most often used with the blockify result marker to compile an expression while saving a text image of the expression (See the SET FILTER condition and the INDEX key expression in STD.CH).

- **Smart stringify result marker:** Stringifies matched input text only if source text is enclosed in parentheses. If no input text matched, it writes nothing to the result text. If the matched input text is a list matched by a list match marker, this result marker stringifies each element in the list (using the same stringify rule) and writes it to the result text.

The smart stringify result marker is designed specifically to support extended expressions for commands other than SETs with <xlToggle> arguments. Extended expressions are command syntax elements that can be specified as literal text or as an expression if enclosed in parentheses. The <xcDatabase> argument of the USE command is a typical example. For instance, if the matched input for the <xcDatabase> argument is the word Customer, it is written to the result text as the string "Customer," but the expression (cPath + cDatafile) would be written to the result text unchanged (i.e., without quotes).

- **Blockify result marker:** Writes matched input text as a code block without any arguments to the result text. For example, the input text x + 3 would be written to the result text as {|| x + 3}. If no input text is matched, it writes nothing to the result text. If the matched input text is a list matched by a list match marker, this result marker blockifies each element in the list.

  The blockify result marker used with the regular and list match markers matches various kinds of expressions and writes them as code blocks to the result text. Remember that a code block is a piece of compiled code to execute sometime later. This is important when defining commands that evaluate expressions more than once per invocation. When defining a command, you can use code blocks to pass an expression to a function and procedure as data rather than as the result of an evaluation. This allows the target routine to evaluate the expression whenever necessary.

  In STD.CH, the blockify result marker defines database commands where an expression is evaluated for each record. Commonly, these are field or expression lists, FOR and WHILE conditions, or key expressions for commands that perform actions based on key values.

- **Logify result marker:** Writes true (.T.) to the result text if any input text is matched; otherwise, it writes false (.F.) to the result text. This result marker does not write the input text itself to the result text.

  The logify result marker is generally used with the restricted match marker to write true (.T.) to the result text if an optional clause is specified with no argument; otherwise it writes false (.F.). In STD.CH, this formulation defines the EXCLUSIVE and SHARED clauses of the USE command.

. **Repeating result clauses** are portions of the <resultPattern> enclosed by square brackets ([ ]). The text within a repeating clause is written to the result text as many times as it has input text for any or all result markers within the clause. If there is no matching input text, the repeating clause is not written to the result text. Repeating clauses, however, cannot be nested. If you need to nest repeating clauses, you probably need an additional #command rule for the current command.

  Repeating clauses are the result pattern part of the #command facility that create optional clauses which have arguments. You can match input text with any match marker other than the restricted match marker and write to the result text with any of the corresponding result markers. Typical examples of this facility are the definitions for the STORE and REPLACE commands in STD.CH.

Notes

. **Less than operator:** If you specify the less than operator (<) in the <resultPattern> expression, you must precede it with the escape character (\).

. **Multistatement lines:** You can specify more than one statement

as a part of the result pattern by separating each statement with a semicolon. If you specify adjacent statements on two separate lines, the first statement must be followed by two semicolons.

Examples

These examples encompass many of the basic techniques you can use when defining commands with the #command and #translate directives. In general, these examples are based on standard commands defined in STD.CH. Note, however, the functions specified in the example result patterns are not the actual functions found in STD.CH, but fictitious functions specified for illustration only.

. This example defines the @...BOX command using regular match markers with regular result markers:

```
#command  @ <top>, <left>, <bottom>, <right> BOX ;
      <boxstring>;
=>;
      CmdBox( <top>, <left>, <bottom>, ;
      <right>,<boxstring> )
```

. This example uses a list match marker with a regular result marker to define the ? command:

```
#command ? [<list,...>] => QOUT(<list>)
```

. This example uses a restricted match marker with a logify result marker to implement an optional clause for a command definition. In this example, if the ADDITIVE clause is specified, the logify result marker writes true (.T.) to the result text; otherwise, it writes false (.F.):

```
#command RESTORE FROM <file> [<add: ADDITIVE>];
=>;
      CmdRestore( <(file)>, <.add.> )
```

. This example uses a list match marker with a smart stringify result marker to write to the result text the list of fields specified as the argument of a FIELDS clause. In this example, the field list is written as an array with each field name as an element of the array:

```
#command COPY TO <file> [FIELDS <fields,...>];
=>;
      CmdCopyAll( <(file)>, { <(fields)> } )
```

. These examples use the wild match marker to define a command that writes nothing to the result text. Do this when attempting to compile unmodified code developed in another dialect:

```
#command SET ECHO <*text*>    =>
#command SET TALK <*text*>    =>
```

. These examples use wild match markers with dumb stringify result markers to match command arguments specified as literals, then write them to the result text as strings in all cases:

```
#command SET PATH TO <*path*>  =>  ;
   SET( _SET_PATH, #<path> )
#command SET COLOR TO <*spec*> =>  SETCOLOR( #<spec> )
```

. These examples use a normal result marker with the blockify result marker to both compile an expression and save the text version of it for later use:

```
#command SET FILTER TO <xpr>;
=>;
```

```
      CmdSetFilter( <{xpr}>, <"xpr"> )

#command INDEX ON <key> TO <file>;
=>;
      CmdCreateIndex( <(File)>, <"Key">, <{Key}> )
```

. This example demonstrates how the smart stringify result marker implements a portion of the USE command for those arguments that can be specified as extended expressions:

```
#command USE <db> [ALIAS <a>];
=>;
      CmdOpenDbf( <(db)>, <(a)> )
```

. This example illustrates the importance of the blockify result marker for defining a database command. Here, the FOR and WHILE conditions matched in the input text are written to the result text as code blocks:

```
#command COUNT [TO <var>];
      [FOR <for>] [WHILE <while>];
      [NEXT <next>] [RECORD <rec>] [<rest:REST>] [ALL];
=>;
      <var> := 0,;
      DBEVAL( {|| <var>++}, <{for}>, <{while}>,;
         <next>, <rec>, <.rest.> )
```

. In this example the USE command again demonstrates the types of optional clauses with keywords in the match pattern; one clause is a keyword followed by a command argument, and the second is solely a keyword:

```
#command USE <db> [<new: NEW>] [ALIAS <a>] ;
      [INDEX <index,...>][<ex: EXCLUSIVE>] ;
      [<sh: SHARED>] [<ro: READONLY>];
=>;
      CmdOpenDbf(<(db)>, <(a)>, <.new.>,;
         IF(<.sh.> .OR. <.ex.>, !<.ex.>, NIL),;
            <.ro.>, {<(index)>})
```

. This example uses the STORE command definition to illustrate the relationship between an optional match clause and a repeating result clause:

```
#command STORE <value> TO <var1> [, <varN> ];
=>;
      <var1> := [ <varN> := ] <value>
```

. This example uses #translate to define a pseudofunction:

```
#translate AllTrim(<cString>) => LTRIM(RTRIM(<cString>))
```

See Also: #define #xcommand

#define
Define a manifest constant or pseudofunction

Syntax

#define <idConstant> [<resultText>]
#define <idFunction>([<arg list>]) [<exp>]

Arguments

<idConstant> is the name of an identifier to define.

<resultText> is the optional replacement text to substitute whenever a valid <idConstant> is encountered.

<idFunction> is a pseudofunction definition with an optional argument list (<arg list>). If you include <arg list>, it is delimited by parentheses (()) immediately following <idFunction>.

<exp> is the replacement expression to substitute when the pseudofunction is encountered. Enclose this expression in parentheses to guarantee precedence of evaluation when the pseudofunction is expanded.

Note: #define identifiers are case-sensitive, where #command and #translate identifiers are not.

Description

The #define directive defines an identifier and, optionally, associates a text replacement string. If specified, replacement text operates much like the search and replace operation of a text editor. As each source line from a program file is processed by the preprocessor, the line is scanned for identifiers. If a currently defined identifier is encountered, the replacement text is substituted in its place.

Identifiers specified with #define follow most of the identifier naming rules in CA-Clipper . Defined identifiers can contain any combination of alphabetic and numeric characters, including underscores. Defined identifiers, however, differ from other identifiers by being case-sensitive. As a convention, defined identifiers are specified in uppercase to distinguish them from other identifiers used within a program. Additionally, identifiers are specified with a one or two letter prefix to group similar identifiers together and guarantee uniqueness. Refer to one of the supplied header files in the \CLIPPER5\INCLUDE directory for examples.

When specified, each definition must occur on a line by itself. Unlike statements, more than one directive cannot be specified on the same source line. You may continue a definition on a subsequent line by employing a semicolon (;). Each #define directive is specified followed by one or more white space characters (spaces or tabs), a unique identifier, and optional replacement text. Definitions can be nested, allowing one identifier to define another.

A defined identifier has lexical scope like a filewide static variable. It is only valid in the program (.prg) file in which it is defined unless defined in STD.CH or the header file specified on the compiler command line with the /U option. Unlike a filewide static variable, a defined identifier is visible from the point where it is defined in the program file until it is either undefined, redefined, or the end of the program file is reached.

You can redefine or undefine existing identifiers. To redefine an identifier, specify a new #define directive with the identifier and the new replacement text as its arguments. The current definition is then overwritten with the new definition, and a compiler warning is issued in

case the redefinition is inadvertent. To undefine an identifier, specify an #undef directive with the identifier as its argument. #define directives have three basic purposes:

- To define a control identifier for #ifdef and #ifndef
- To define a manifest constant--an identifier defined to represent a constant value
- To define a compiler pseudofunction

The following discussion expands these three purposes of the #define directive in your program.

Preprocessor Identifiers

The most basic #define directive defines an identifier with no replacement text. You can use this type of identifier when you need to test for the existence of an identifier with either the #ifdef or #ifndef directives. This is useful to either exclude or include code for conditional compilation. This type of identifier can also be defined using the /D compiler option from the compiler command line. See the examples below.

Manifest Constants

The second form of the #define directive assigns a name to a constant value. This form of identifier is referred to as a manifest constant. For example, you can define a manifest constant for the INKEY() code associated with a key press:

```
#define K_ESC  27
IF LASTKEY() = K_ESC
   .
   . <statements>
   .
ENDIF
```

Whenever the preprocessor encounters a manifest constant while scanning a source line, it replaces it with the specified replacement text.

Although you can accomplish this by defining a variable, there are several advantages to using a manifest constant: the compiler generates faster and more compact code for constants than for variables; variables have memory overhead where manifest constants have no runtime overhead, thus saving memory and increasing execution speed; using a variable to represent a constant value is conceptually inconsistent. A variable by nature changes and a constant does not.

Use a manifest constant instead of a constant for several reasons. First, it increases readability. In the example above, the manifest constant indicates more clearly the key being represented than does the INKEY() code itself. Second, manifest constants localize the definition of constant values, thereby making changes easier to make, and increasing reliability. Third, and a side effect of the second reason, is that manifest constants isolate implementation or environment specifics when they are represented by constant values.

To further isolate the effects of change, manifest constants and other identifiers can be grouped together into header files allowing you to share identifiers between program (.prg) files, applications, and groups of programmers. Using this methodology, definitions can be standardized for use throughout a development organization. Merge header files into the current program file by using the #include directive.

For examples of header files, refer to the supplied header files in the \CLIPPER5\INCLUDE directory.

Compiler Pseudofunctions

In addition to defining constants as values, the #define directive can also define pseudofunctions that are resolved at compile time. A pseudofunction definition is an identifier immediately followed by an argument list, delimited by parentheses, and the replacement expression. For example:

```
#define AREA(nLength, nWidth)  (nLength * nWidth)
#define SETVAR(x, y)           (x := y)
#define MAX(x, y)              (IF(x > y, x, y))
```

Pseudofunctions differ from manifest constants by supporting arguments. Whenever the preprocessor scans a source line and encounters a function call that matches the pseudofunction definition, it substitutes the function call with the replacement expression. The arguments of the function call are transported into the replacement expression by the names specified in the argument list of the identifier definition. When the replacement expression is substituted for the pseudofunction, names in the replacement expression are replaced with argument text. For example, the following invocations:

```
? AREA(10, 12)
SETVAR(nValue, 10)
? MAX(10, 9)
```

are replaced by :

```
? (10 * 12)
nValue := 10
? (IF(10 > 9, 10, 9)
```

It is important when defining pseudofunctions, that you enclose the result expression in parentheses to enforce the proper order of evaluation. This is particularly important for numeric expressions. In pseudofunctions, you must specify all arguments. If the arguments are not specified, the function call is not expanded as a pseudofunction and exits the preprocessor to the compiler as encountered.

Pseudofunctions do not entail the overhead of a function call and are therefore generally faster. They also use less memory. Pseudofunctions, however, are more difficult to debug within the debugger, have a scope different from declared functions and procedures, do not allow skipped arguments, and are case-sensitive.

You can avoid some of these deficiencies by defining a pseudofunction using the #translate directive. #translate pseudofunctions are not case-sensitive, allow optional arguments, and obey the dBASE four-letter rule. See the #translate directive reference in this chapter for more information.

Examples

. In this example a manifest constant conditionally controls the compilation of debugging code:

```
#define DEBUG
.
. <statements>
.
#ifdef DEBUG
   Assert(FILE("System.dbf"))
#endif
```

. This example defines a manifest constant and substitutes it for an INKEY() value:

```
#define K_ESC  27
```

```
.
. <statements>
.
IF INKEY() != K_ESC
   DoIt()
ELSE
   StopIt()
ENDIF
```

. This example defines pseudofunctions for the standard CA-Clipper functions, MAX() and ALLTRIM():

```
#define MAX(arg1, arg2)   (IF(arg1 > arg2, ;
   arg1, arg2))
#define ALLTRIM(cString)  (RTRIM(LTRIM(cString)))
.
. <statements>
.
? MAX(1, 2)
? ALLTRIM("  Hello  ")
```

See Also: #command #ifdef #ifndef #undef #xcommand

#error
Generate a compiler error and display a message

Syntax

#error [<messageText>]

Arguments

<messageText> is the text of the message to be displayed.
<messageText> is a literal character string--do not enclose the message in quotations unless you want them to appear as part of the display.

Description

#error causes the compiler to generate error number C2074. If the <messageText> parameter is specified, an error message is displayed.

Examples

. This example displays an error message based on whether or not a NETWORK identifier was defined:

```
#ifdef NETWORK
   #error Network version not implemented.
#endif
```

See Also: #command #ifdef #ifndef

#ifdef
Compile a section of code if an identifier is defined

Syntax

```
#ifdef <identifier>
   <statements>...
[#else]
   <statements>...
#endif
```

Arguments

<identifier> is the name of a definition whose existence is being verified.

Description

#ifdef...#endif lets you perform a conditional compilation. It does this by identifying a section of source code to be compiled if the specified <identifier> is defined. The <identifier> can be defined using either the #define directive or the /D compiler option which lets you define an identifier or manifest constant from the compiler command line.

The #else directive specifies the code to compile if <identifier> is undefined. The #endif terminates the conditional compilation block.

Conditional compilation is particularly useful when maintaining many different versions of the same program. For example the demo code and full system code could be included in the same program file and controlled by a single #define statement.

Examples

. This code fragment is a general skeleton for conditional compilation with #ifdef:

```
#define DEMO
.
. <statements>
.
#ifdef DEMO
   <demo specific statements>
#endif
```

. This example controls conditional compilation with an identifier defined on the compiler command line with the /D option:

In DOS:

```
C>CLIPPER Myfile /DDEBUG
```

In the program (.prg) file:

```
#ifdef DEBUG
   Assert(<some condition>)
#endif
```

. This example defines a manifest constant to one value if it does not exist and redefines it to another if it exists:

```
#ifdef M_MARGIN
   #undef  M_MARGIN
   #define M_MARGIN  15
#else
   #define M_MARGIN  10
#endif
```

See Also: #define #ifndef

#ifndef
Compile a section of code if an identifier is undefined

Syntax

```
#ifndef <identifier>
   <statements>...
[#else]
   <statements>...
#endif
```

Arguments

<identifier> is the name of a definition whose absence is being verified.

Description

#ifndef...#endif lets you perform conditional compilation by identifying a section of source code to compile if the specified <identifier> is undefined.

The #else directive specifies the code to compile if <identifier> is defined. The #endif terminates the conditional compilation block.

Examples

. This code fragment is a general skeleton for conditional compilation with #ifndef:

```
#define DEBUG
.
. <statements>
.
#ifndef DEBUG
   <optimized version of code>
#else
   <debugging version of code>
#endif
```

. This example compiles a section of code if a specific identifier is undefined.

In DOS:

```
C>CLIPPER Myfile
```

In the program (.prg) file:

```
#ifndef NODEBUG
   Assert(<some condition>)
#endif
```

. This example overrides a default definition in the program (.prg) file using a manifest constant defined on the compiler command line with the /D option:

In DOS:

```
C>CLIPPER Myfile /DM_MARGIN=10
```

In the program (.prg) file:

```
#ifndef M_MARGIN
   #define M_MARGIN   15
#endif
```

See Also: #define #ifdef

#include
Include a file into the current source file

Syntax

#include "<headerFileSpec>"

Arguments

<headerFileSpec> specifies the name of another source file to include in the current source file. As indicated in the syntax, the name must be enclosed in double quotation marks.

<headerFileSpec> may contain an explicit path and filename as well as a file extension. If, however, no path is specified, the preprocessor searches the following places:

. Source file directory

. Directories supplied with the /I option

. Directories specified in the INCLUDE environment variable

#include directives may be nested up to 15 levels deep--that is, a file that has been included may contain #include directives, up to 15 levels.

Description

#include inserts the contents of the specified file in place of the #include directive in the source file. By convention, the file inserted is referred to as a header file. Header files should contain only preprocessor directives and external declarations. By convention CA-Clipper header files have a (.ch) extension.

When deciding where to locate your header files, you have two basic choices. You can place them in the source file directory where they are local to the current system. Or, you can make them globally available by placing them in the directory specified in the INCLUDE environment variable. A list of one or more directories can be specified.

Header files overcome the one major drawback of defining constants or inline functions--the #define directive only affects the file in which it is contained. This means that every program which needs access to these statements must have a list of directives at the top. The solution to this problem is to place #define statements in a separate file and use the #include directive to tell the preprocessor to include that file before compiling.

For example, suppose the file "Inkey.ch" contains a list of #define directives assigning key values to constants. Instead of including these directives at the top of each program file (.prg) requiring access to them, you can simply place the following line at the top of each program file:

#include "Inkey.ch"

This causes the preprocessor to look for Inkey.ch and place all the directives contained within it at the top of this program.

Another advantage of using the #include directive is that all the #define statements are contained in one file. If any modifications to these statements are necessary, only the #include file need be altered; the program itself remains untouched.

Note that the scope of definitions within an included header file is the current program file unless the header file is included on the compiler command line with the /U option. In this case, the scope is all the

program files compiled in the current invocation of the compiler.

Notes

- **Supplied header files:** CA-Clipper provides a number of header files containing manifest constants for common operations. Refer to \CLIPPER5\INCLUDE for more information.

- **STD.CH--the standard header file:** STD.CH is the standard header file provided with CA-Clipper. Its default location is \CLIPPER5\INCLUDE. STD.CH contains the definitions of all CA-Clipper commands and the standard functions specified as pseudofunctions. It is strongly recommended that no changes be made to STD.CH. If changes are desired, it is advisable to copy STD.CH to a new name, make the changes, and compile with /U.

  This header file differs somewhat from a header file you might #include in that everything defined in STD.CH, with #define, #translate, or #command, has a scope of the entire compile rather than the current source file.

Examples

- This example uses #include to insert Inkey.ch, a file of common keyboard definitions, into a key exception handler called by an interface function:

```
#include "Inkey.ch"

FUNCTION GetEvent()
   LOCAL nKey, nResult
   nKey = INKEY(0)
   DO CASE
   CASE nKey = K_F10
      nResult := DoMenu("Browse")
   CASE nKey = K_ESC
      nResult := DoQuit()
      .
      . <statements>
      .
   CASE nKey = K_CTRL_RIGHT
      nResult := DoNextRec()
   ENDCASE
   RETURN nResult
```

See Also: #command #define

#stdout
Send literal text to the standard output device

Syntax

#stdout [<messageText>]

Arguments

<messageText> is the text of the message to display. <messageTest> is a literal character string. Do not enclose the message in quotation marks unless you want them to appear as part of the display.

Description

#stdout causes the compiler to output the literal text to the standard output device (stdout) during compilation. If <messageText> is not specified, a carriage return/line feed pair echoes to stdout.

Warning! Manifest constants are not translated in #stdout. Implementation is identical to #error with the following exceptions: output is written to STDOUT and no compiler error is generated.

Examples

. This example demonstrates use of #stdout:

```
#ifdef DEBUG
   #stdout Compiling debugging version...
#endif

PROCEDURE Main()

   ? "Hello world"
   RETURN

   #stdout End of "Hello World" program
```

See Also: #error

#undef
Remove a #define macro definition

Syntax

#undef <identifier>

Arguments

<identifier> is the name of the manifest constant or pseudofunction to remove.

Description

#undef removes an identifier defined with the #define directive. After an #undef, the specified dentifier becomes undefined. Use #undef to remove an identifier before you redefine it with #define, preventing the compiler warning that occurs when an existing identifier is redefined. Also, use #undef to make conditional compilation specific to certain sections of a program.

Examples

. To define then undefine a manifest constant and a pseudofunction:

```
#define K_ESC  27
#define MAX(x, y)  IF(x > y, x, y)
.
. <statements>
.
#undef K_ESC
#undef MAX
```

. To use #undef to undefine an identifier before redefining it:

```
#define DEBUG
.
. <statements>
.
#undef DEBUG
#define DEBUG  .T.
```

. To undefine an identifier if it exists, and otherwise define it for later portions of the program file:

```
#ifdef TEST
   #undef TEST
#else
   #define TEST
#endif
```

See Also: #define #ifdef #ifndef

#xcommand | #xtranslate
Specify a user-defined command or translation directive

Syntax

```
#xcommand   <matchPattern> => <resultPattern>
#xtranslate <matchPattern> => <resultPattern>
```

Arguments

<matchPattern> is the pattern to match in the input text.

<resultPattern> is the text produced if a piece of input text matches the <matchPattern>.

Description

The #xcommand and #xtranslate directives work like #command and #translate except that they overcome the dBASE keyword length limitation. They are significant beyond the first four letters, limited only by available memory. All other rules apply.

See Also: #command

Description

Getsys.prg, the standard GET/READ system, has been restructured to provide a cleaner platform for extensions.

Some of the functions in the Getsys.prg have been made public so that they can be used when implementing customized Get readers. These functions are described below.

GETACTIVE()
Return the currently active Get object

Syntax

GETACTIVE() --> objGet

Returns

GETACTIVE() returns the current active Get object within the current READ. If there is no READ active when GETACTIVE() is called, it returns NIL.

Description

GETACTIVE() is an environment function that provides access to the active GET object during a READ. The current active Get object is the one with input focus at the time GETACTIVE() is called.

Examples

. This code uses a WHEN clause to force control to branch to a special reader function. Within this function, GETACTIVE() retrieves the active Get object:

```
@ 10, 10 GET x
@ 11, 10 GET y WHEN MyReader()
@ 12, 10 GET z
READ

// Called just before second get (above)
// becomes current
FUNCTION MyReader
   LOCAL objGet               // Active Get holder
   objGet := GETACTIVE()      // Retrieve current
                              // active Get
   BarCodeRead( objGet )
   RETURN (.F.)               // Causes Get to be
                              // skipped in READ
```

Files: Library is CLIPPER.LIB, source file is Getsys.prg.

See Also: @...SAY READ READMODAL()

GETAPPLYKEY()
Apply a key to a Get object from within a Get reader

Syntax

GETAPPLYKEY(<oGet>, <nKey>) --> NIL

Arguments

<oGet> is a reference to a Get object.

<nKey> is the INKEY() value to apply to <oGet>.

Returns

GETAPPLYKEY() always returns NIL.

Description

GETAPPLYKEY() is a Get system function that applies an INKEY() value to a Get object. Keys are applied in the default way. That is, cursor movement keys change the cursor position within the Get, data keys are entered into the Get, etc.

If the key supplied to GETAPPLYKEY() is a SET KEY, GETAPPLYKEY() will execute the set key and return; the key is not applied to the Get object.

Notes

- **Focus:** The Get object must be in focus before keys are applied. Refer to Get:setFocus and Get:killFocus for more information.

- **CLEAR GETS:** The Get object must be in focus before keys are applied. Refer to Get:setFocus and Get:killFocus for more information.

Files: Library is CLIPPER.LIB, source file is Getsys.prg.

See Also: GETPOSTVALIDATE() GETPREVALIDATE() GETREADER() READMODAL()

GETDOSETKEY()
Process SET KEY during Get editing

Syntax

GETDOSETKEY(<oGet>) --> NIL

Arguments

<oGet> is a reference to the current Get object.

Returns

GETDOSETKEY() always returns NIL.

Description

GETDOSETKEY() is a Get system function that executes a SET KEY code block, preserving the context of the passed Get object.

Note that the procedure name and line number passed to the SET KEY block are based on the most recent call to READMODAL().

Notes

. If a CLEAR GETS occurs in the SET KEY code, Get:exitState is set to GE_ESCAPE. In the standard system this cancels the current Get object processing and terminates READMODAL().

Files: Library is CLIPPER.LIB, source file is Getsys.prg.

See Also: READMODAL() SET KEY

GETPOSTVALIDATE()
Postvalidate the current Get object

Syntax

GETPOSTVALIDATE(<oGet>) --> lSuccess

Arguments

<oGet> is a reference the current Get object.

Returns

GETPOSTVALIDATE() returns a logical value indicating whether the Get object has been postvalidated successfully.

Description

GETPOSTVALIDATE() is a Get system function that validates a Get object after editing, including evaluating Get:postBlock (the VALID clause) if present.

The return value indicates whether the Get has been postvalidated successfully. If a CLEAR GETS is issued during postvalidation, Get:exitState is set to GE_ESCAPE and GETPOSTVALIDATE() returns true (.T.).

Notes

. In the default system, a Get:exitState of GE_ESCAPE cancels the current Get and terminates READMODAL().

Files: Library is CLIPPER.LIB, source file is Getsys.prg.

See Also: GETAPPLYKEY() GETPREVALIDATE() GETREADER() READMODAL()

GETPREVALIDATE()
Prevalidate a Get object

Syntax

GETPREVALIDATE(<oGet>) --> lSuccess

Arguments

<oGet> is a reference the current Get object.

Returns

GETPREVALIDATE() returns a logical value indicating whether the Get object has been prevalidated successfully.

Description

GETPREVALIDATE() is a Get system function that validates the Get object for editing, including evaluating Get:preBlock (the WHEN clause) if it is present. The logical return value indicates whether the Get has been prevalidated successfully.

Get:exitState is also set to reflect the outcome of the prevalidation:

Get:exitState Values

Getexit.ch	Meaning
GE_NOEXIT	Indicates prevalidation success, okay to edit
GE_WHEN	Indicates prevalidation failure
GE_ESCAPE	Indicates that a CLEAR GETS was issued

Note that in the default system, a Get:exitState of GE_ESCAPE cancels the current Get and terminates READMODAL().

Files: Library is CLIPPER.LIB, source file is Getsys.prg.

See Also: GETAPPLYKEY() GETPOSTVALIDATE() GETREADER() READMODAL()

GETREADER()
Execute standard READ behavior for a Get object

Syntax

GETREADER(<oGet>) --> NIL

Arguments

<oGet> is a reference to a Get object.

Returns

GETREADER() always returns NIL.

Description

GETREADER() is a Get system function that implements the standard READ behavior for Gets. By default, READMODAL() uses the GETREADER() function to read Get objects. GETREADER() in turn uses other functions in Getsys.prg to do the work of reading the Get object.

Notes

. If a Get object's Get:reader instance variable contains a code block, READMODAL() will evaluate that block in lieu of the call to GETREADER(). For more information refer to the Get:reader reference.

Files: Library is CLIPPER.LIB, source file is Getsys.prg.

See Also: GETAPPLYKEY() GETPOSTVALIDATE() GETPREVALIDATE() READMODAL()

READFORMAT()
Return, and optionally set, the code block that implements a format (.fmt) file

---

Syntax

READFORMAT([<bFormat>]) --> bCurrentFormat

Arguments

<bFormat> is the name of the code block, if any, to use for implementing a format file. If no argument is specified, the function simply returns the current code block without setting a new one.

Returns

READFORMAT() returns the current format file as a code block. If no format file has been set, READFORMAT() returns NIL.

Description

READFORMAT() is a Get system function that accesses the current format file in its internal code block representation. It lets you manipulate the format file code block from outside of the Get system source code.

To set a format file, use SET FORMAT (see the SET FORMAT entry in the Reference guide) or READFORMAT().

READFORMAT() is intended primarily for creating new READ Layers. The code block that READFORMAT() returns, when evaluated, executes the code that is the format file from which it was created.

Files: Library is CLIPPER.LIB, source file is Getsys.prg.

See Also: READKILL() READUPDATED() SET FORMAT*

ASIZE()
Grow or shrink an array

Syntax

ASIZE(<aTarget>, <nLength>) --> aTarget

Arguments

<aTarget> is the array to grow or shrink.

<nLength> is the new size of the array.

Returns

ASIZE() returns a reference to the target array, <aTarget>.

Description

ASIZE() is an array function that changes the actual length of the <aTarget> array. The array is shortened or lengthened to match the specified length. If the array is shortened, elements at the end of the array are lost. If the array is lengthened, new elements are added to the end of the array and assigned NIL.

ASIZE() is similar to AADD() which adds a single new element to the end of an array and optionally assigns a new value at the same time. Note that ASIZE() is different from AINS() and ADEL(), which do not actually change the array's length.

Examples

. These examples demonstrate adding new elements and deleting existing elements:

```
aArray := { 1 }          // Result: aArray is { 1 }
ASIZE(aArray, 3)         // Result: aArray is { 1, NIL, NIL }
ASIZE(aArray, 1)         // Result: aArray is { 1 }
```

Files: Library is CLIPPER.LIB.

See Also: AADD() ADEL() AFILL() AINS()

READKILL()
Return, and optionally set, whether the current READ should be exited

---

Syntax

READKILL([<lKillRead>]) --> lCurrentSetting

Arguments

<lKillRead> sets the READKILL() flag. A value of true (.T.) indicates that the current read should be terminated, and a value of false (.F.) indicates that it should not.

Returns

READKILL() returns the current setting as a logical value.

Description

READKILL() is a Get system function that lets you control whether or not to terminate the current READ.

Unless directly manipulated, READKILL() returns true (.T.) after you issue a CLEAR GETS (see the CLEAR GETS entry in the Reference guide) for the current READ and otherwise returns false (.F.).

By accessing the function directly, however, you can control the READKILL() flag with its function argument and use it to create new READ Layers.

Files: Library is CLIPPER.LIB, source file is Getsys.prg.

See Also: CLEAR GETS READFORMAT() READUPDATED()

READUPDATED()
Return, and optionally set, whether any Get changed during a READ

Syntax

READUPDATED([<lChanged>]) --> lCurrentSetting

Arguments

<lChanged> sets the READUPDATED() flag...A value of true (.T.) indicates that data has changed, and a value of false (.F.) indicates that no change has occurred.

Returns

READUPDATED() returns the current setting as a logical value.

Description

READUPDATED() is a Get system function intended primarily for creating new READ Layers. It is identical in functionality to UPDATED() (see the UPDATED() entry in the Reference guide), except that it allows the UPDATED() flag to be set.

READUPDATED() enables you to manipulate the UPDATED() flag from outside of the Get system source code.

Files: Library is CLIPPER.LIB, source file is Getsys.prg.

See Also: READFORMAT() READKILL() UPDATED()

CA-Clipper 5.2 provides many new and enhanced commands and functions that can be used to access and manipulate databases and get specific information about the RDD in use. These language elements were summarized in the previous chapter, and this chapter provides a detailed alphabetical reference to them.

This chapter contains entries for all new CA-Clipper commands and functions designed for use with RDDs. As for existing language elements, only those that were changed to accommodate the new RDD architecture are documented here.

Other language elements that are specifically designed for database manipulation and have not been altered (e.g., USE) are documented in the Reference guide. Using Appendix G: Categorized Language Tables in the Error Messages and Appendices guide, you can find these language elements categorized under database or index.

The refinement of the CA-Clipper language, both additions and enhancements, ensures consistent language support for access to different data formats and allows you to design applications that are not dependent on a particular data format. In other words, you can use the same commands and functions to manipulate a FoxPro database as you would use to manipulate a dBASE IV database.

The entries in this chapter are organized like the entries in the Reference guide. The **Status** section, however, is new to this chapter. This section declares the command or function as New or Enhanced. These categories are defined as follows:

- **New:** The entry is completely new, as of version 5.2, to the CA-Clipper language.

- **Enhanced:** The entry is an established part of the CA-Clipper language (i.e., it existed prior to version 5.2) but has been modified to accommodate the new RDD architecture.

  All syntax changes for enhanced commands and functions are marked in **bold**.

APPEND FROM command
Import records from a (.dbf) or ASCII file

Syntax

```
APPEND FROM <xcFile>
   [FIELDS <idField list>]
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
   [SDF | DELIMITED [WITH BLANK | <xcDelimiter>] |
   [VIA <xcDriver>]]
```

Arguments

FROM <xcFile> specifies the name of the source file. You can specify <xcFile> either as a literal filename or as a character expression enclosed in parentheses. If a file extension is not specified, (.dbf) is the default input file type. If SDF or DELIMITED is specified, the file extension is assumed to be (.txt) unless otherwise specified.

FIELDS <idField list> specifies the list of fields to copy from <xcFile>. The default is all fields.

<scope> is the portion of the source database file to APPEND FROM. NEXT <n> APPENDs the first <n> records. RECORD <n> APPENDs only record number <n> from <xcFile>. The default scope is ALL records in <xcFile>.

WHILE <lCondition> specifies the set of records meeting the condition from the first record in the source file until the condition fails.

FOR <lCondition> specifies the conditional set of records to APPEND FROM within the given scope.

SDF identifies a System Data Format ASCII file. Records and fields are fixed length.

DELIMITED identifies an ASCII text file, where character fields are enclosed in double quote marks (the default delimiter). Note that delimiters are not required and CA-Clipper correctly APPENDs character fields not enclosed in them. Fields and records are variable length.

DELIMITED WITH BLANK identifies an ASCII text file in which fields are separated by one space and character fields are not enclosed in delimiters.

DELIMITED WITH <xcDelimiter> identifies a delimited ASCII text file, where character fields are enclosed using the specified delimiter. You can specify <xcDelimiter> as a literal character or as a character expression enclosed in parentheses.

See the tables below for more information regarding the format specification requirements for ASCII text files that you want to APPEND using these arguments.

VIA <xcDriver> specifies the replaceable database driver (RDD) to use to import the desired data. <cDriver> is name of the RDD specified as a character expression. If <cDriver> is specified as a literal value, it must be enclosed in quotes.

If the VIA clause is omitted, APPEND FROM uses the driver in the current work area. If you specify the VIA clause, you must REQUEST the appropriate RDDs be linked into the application.

Warning! If the DELIMITED WITH clause is specified on a COPY TO or APPEND FROM command line, it must be the last clause specified.

Description

APPEND FROM adds records to the current database file from an ASCII text file or another database file. Only fields with the same names and types are APPENDed. Fields with the same name from both the current database file and <xcFile> must be the same data type. If they are not, a runtime error occurs when the APPEND FROM command is invoked.

Any date information in <xcFile> must be in the format yyyymmdd to be properly APPENDed.

In a network environment, APPEND FROM does not require that the current database file be USEed EXCLUSIVEly or locked with FLOCK() to perform its operation. As each record is added, CA-Clipper automatically arbitrates contention for the new record.

When you invoke APPEND FROM, CA-Clipper attempts to open <xcFile> shared and read-only. If access is denied, APPEND FROM terminates with a runtime error. Refer to the Network Programming chapter in the Programming and Utilities guide for more information. No error is raised if you attempt to open a (.dbf) file that is already open.

This table shows the format specifications for SDF text files.

SDF Text File Format Specifications

File Element	Format
Character fields	Padded with trailing blanks
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Padded with leading blanks or zeros
Field separator	None
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

This table shows the format specifications for DELIMITED and DELIMITED WITH <xcDelimiter> ASCII text files.

DELIMITED Text File Format Specifications

File Element	Format
Character fields	May be delimited, with trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros may be truncated
Field separator	Comma
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

This table shows the format specifications for DELIMITED WITH BLANK ASCII text files.

DELIMITED WITH BLANK Text File Format Specifications

File Element	Format
Character fields	Not delimited, trailing blanks may be truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros may be truncated
Field separator	Single blank space

```
Record separator    Carriage return/linefeed
End of file marker  1A hex or CHR(26)
------------------------------------------------------------------------
```

Notes

- **Deleted records:** If DELETED is OFF, deleted records in <xcFile> are APPENDed to the current database file and retain their deleted status. If DELETED is ON, however, none of the deleted <xcFile> records are APPENDed.

- **Unmatched field widths:** If a field in the current database file is character type and has a field length greater than the incoming <xcFile> data, CA-Clipper pads the <xcFile> data with blanks. If the current field is character data type and its field length is less than the incoming <xcFile> data, the <xcFile> data is truncated to fit. If the current field is numeric type and the incoming <xcFile> data has more digits than the current field length, a runtime error occurs.

Examples

- This example demonstrates an APPEND FROM command using a fields list and a condition:

```
USE Sales NEW
APPEND FROM BranchFile FIELDS Branch, Salesman, Amount;
   FOR Branch = 100
```

- This example demonstrates how a <scope> can be specified to import a particular record from another database file:

```
APPEND RECORD 5 FROM Temp
```

Files: Library is CLIPPER.LIB.

Status: Enhanced

See also: COPY TO

See Also: COPY TO

COPY TO command
Export records to a new (.dbf) or ASCII file

Syntax

```
COPY [FIELDS <idField list>] TO <xcFile>
   [<scope>] [WHILE <lCondition>] [FOR <lCondition>]
   [SDF | DELIMITED [WITH BLANK | <xcDelimiter>] |
   [VIA <xcDriver>]]
```

Arguments

FIELDS <idField list> specifies the list of fields to copy to the target file. The default is all fields.

TO <xcFile> specifies the name of the target file. The filename can be specified either as a literal filename or as a character expression enclosed in parentheses. If SDF or DELIMITED is specified, (.txt) is the default extension. Otherwise, (.dbf) is the default extension.

<scope> defines the portion of the current database file to COPY. The default is ALL records.

WHILE <lCondition> specifies the set of records meeting the condition from the current record until the condition fails.

FOR <lCondition> specifies the conditional set of records to COPY within the given scope.

SDF specifies the output file type as a System Data Format ASCII text file. Records and fields are fixed length.

DELIMITED specifies the output file type as a delimited ASCII text file where character fields are enclosed in double quote marks (the default delimiter). Records and fields are variable length.

DELIMITED WITH BLANK identifies an ASCII text file in which fields are separated by one space and character fields have no delimiters.

DELIMITED WITH <xcDelimiter> identifies a delimited ASCII text file where character fields are enclosed using the specified delimiter. <xcDelimiter> can be specified either as a literal character or as a character expression enclosed in parentheses.

See the tables below for more information regarding the format specifications for ASCII text files created using these arguments.

VIA <xcDriver> specifies the replaceable database driver (RDD) to use to create the resulting copy. <cDriver> is name of the RDD specified as a character expression. If <cDriver> is specified as a literal value, it must be enclosed in quotes.

If the VIA clause is omitted, COPY TO uses the driver in the current work area. If you specify the VIA clause, you must REQUEST the appropriate RDDs be linked into the application.

Note: If the DELIMITED WITH clause is specified on a COPY or APPEND command, it must be the last clause specified.

Description

COPY TO is a database command that copies all or part of the current database file to a new file. Records contained in the active database file are copied unless limited by a <scope>, a FOR|WHILE clause, or a filter.

If DELETED is OFF, deleted records in the source file are copied to

<xcFile> where they retain their deleted status. If DELETED is ON, however, no deleted records are copied. Similarly, if a FILTER has been SET invisible records are not copied.

Records are copied in controlling index order if there is an index open in the current work area and SET ORDER is not zero. Otherwise, records are copied in natural order.

In a network environment, CA-Clipper opens the target database file EXCLUSIVEly before the COPY TO operation begins. Refer to the Network Programming chapter in the Programming and Utilities guide for more information.

This table shows the format specifications for SDF text files.

SDF Text File Format Specifications

File Element	Format
Character fields	Padded with trailing blanks
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Padded with leading blanks for zeros
Field separator	None
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

This table shows the format specifications for DELIMITED and DELIMITED WITH <xcDelimiter> ASCII text files.

DELIMITED Text File Format Specifications

File Element	Format
Character fields	Delimited, with trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros truncated
Field separator	Comma
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

This table shows the format specifications for DELIMITED WITH BLANK ASCII text files.

DELIMITED WITH BLANK Text File Format Specifications

File Element	Format
Character fields	Not delimited, trailing blanks truncated
Date fields	yyyymmdd
Logical fields	T or F
Memo fields	Ignored
Numeric fields	Leading zeros truncated
Field separator	Single blank space
Record separator	Carriage return/linefeed
End of file marker	1A hex or CHR(26)

Examples

. This example demonstrates COPYing to another database file:

```
USE Sales NEW
```

```
COPY TO Temp
```

. This example demonstrates the layout of an SDF file with four fields, one for each data type:

```
USE Testdata NEW
COPY NEXT 1 TO Temp SDF
TYPE Temp.txt

Result:

Character     12.0019890801T
```

. This example demonstrates the layout of a DELIMITED file:

```
COPY NEXT 1 TO Temp DELIMITED
TYPE Temp.txt

Result:

"Character",12.00,19890801,T
```

. This example demonstrates the layout of a DELIMITED file WITH a different delimiter:

```
COPY NEXT 1 TO Temp DELIMITED WITH '
TYPE Temp.txt

Result:

'Character',12.00,19890801,T
```

Files: Library is CLIPPER.LIB.

Status: Enhanced

See also: APPEND FROM, COPY FILE, COPY STRUCTURE, SET DELETED

See Also: APPEND FROM COPY FILE COPY STRUCTURE SET DELETED

DBAPPEND() function
Append a new record to the database open in the current work area

---

Syntax

DBAPPEND([**<lReleaseRecLocks>**]) --> NIL

Arguments

<lReleaseRecLocks> is a logical data type that if true (.T.), clears all pending record locks, then appends the next record. If <lReleaseRecLocks> is false (.F.), all pending record locks are maintained and the new record is added to the end of the Lock List. The default value of <lReleaseRecLocks> is true (.T.).

Returns

DBAPPEND() always returns NIL.

Description

DBAPPEND() is a database function that lets you add records to the current database. The enhancement to this function lets you maintain multiple record locks during an append.

DBAPPEND(), without a parameter, as in earlier versions of CA-Clipper, clears all pending record locks prior to an append. This is the same as DBAPPEND(.T.).

Examples

. This example appends a blank record to the database, Sales, without releasing the record locks in the current Lock List, then checks for a network error:

```
USE Sales NEW
SET INDEX TO Sales

DBAPPEND(.F.)

IF NETERR()
   ? "A network error has occurred!"
ENDIF
```

Status: Enhanced

See also: APPEND BLANK

See Also: APPEND BLANK

ASORT()
Sort an array

Syntax

ASORT(<aTarget>, [<nStart>],
[<nCount>], [<bOrder>]) --> aTarget

Arguments

<aTarget> is the array to sort.

<nStart> is the first element of the sort. If not specified, the default starting position is one.

<nCount> is the number of elements to sort. If not specified, all elements in the array beginning with the starting element are sorted.

<bOrder> is an optional code block used to determine sorting order. If not specified, the default order is ascending.

Returns

ASORT() returns a reference to the <aTarget> array.

Description

ASORT() is an array function that sorts all or part of an array containing elements of a single data type. Data types that can be sorted include character, date, logical, and numeric.

If the <bOrder> argument is not specified, the default order is ascending. Elements with low values are sorted toward the top of the array (first element), while elements with high values are sorted toward the bottom of the array (last element).

If the <bOrder> block argument is specified, it is used to determine the sorting order. Each time the block is evaluated, two elements from the target array are passed as block parameters. The block must return true (.T.) if the elements are in sorted order. This facility can be used to create a descending or dictionary order sort. See the examples below.

When sorted, character strings are ordered in ASCII sequence; logical values are sorted with false (.F.) as the low value; date values are sorted chronologically; and numeric values are sorted by magnitude.

Note: Because CA-Clipper implements multidimensional arrays by nesting subarrays within other arrays, ASORT() will not directly sort a multidimensional array. To sort a nested array, you must supply a code block which properly handles the subarrays.

Examples

. This example creates an array of five unsorted elements, sorts the array in ascending order, then sorts the array in descending order using a code block:

```
aArray := { 3, 5, 1, 2, 4 }
ASORT(aArray)
// Result: { 1, 2, 3, 4, 5 }

ASORT(aArray,,, { |x, y| x > y })
// Result: { 5, 4, 3, 2, 1 }
```

. This example sorts an array of character strings in ascending order, independent of case. It does this by using a code block that converts the elements to uppercase before they are compared:

```
aArray := { "Fred", Kate", "ALVIN", "friend" }
ASORT(aArray,,, { |x, y| UPPER(x) < UPPER(y) })
```

. This example sorts a nested array using the second element of each subarray:

```
aKids := { {"Mary", 14}, {"Joe", 23}, {"Art", 16} }
aSortKids := ASORT(aKids,,, { |x, y| x[2] < y[2] })

Result:

{ {"Mary", 14}, {"Art", 16}, {"Joe", 23} }
```

Files: Library is EXTEND.LIB.

See Also: ASCAN() EVAL() SORT

DBGOTO() function
Position record pointer to a specific identity

Syntax

DBGOTO(**<xIdentity>**) --> NIL

Arguments

<xIdentity> is a unique value guaranteed by the structure of the data file to reference a specific item in a data source (database). In a (.dbf) <xIdentity> is the record number. In other data formats, <xIdentity> is the unique primary key value. <xIdentity> could be an array offset or virtual memory handle if the data set is in memory instead of on disk.

Returns

DBGOTO() always returns NIL.

Description

DBGOTO() is a database function that positions the record pointer in the current work area at the specified <xIdentity>. In an Xbase data structure, this identity is the record number because every record, even an empty record, has a record number. In non-Xbase data structures, identity may be defined as something other than record number.

Examples

. This example saves the current record number, searches for a key, then restores the record pointer to the saved position:

```
FUNCTION KeyExists( xKeyExpr )

   LOCAL nSavRecord := RECNO()      // Save the current record
                                    // pointer position
   LOCAL lFound

   SEEK xKeyExpr
   IF ( lFound := FOUND() )
   .
   .< statements >
   .
   ENDIF

   DBGOTO( nSavRecord )             // Restore the record
                                    // pointer position
   RETURN ( lFound )
```

Status: Enhanced

See also: BOF(), DBGOBOTTOM(), DBGOTOP(), DBSEEK(), DBSKIP(), EOF(), GO

See Also: BOF() DBGOBOTTOM() DBGOTOP() DBSEEK() DBSKIP()

DBRLOCK() function
Lock the record at the current or specified identity

Syntax

```
DBRLOCK([<xIdentity>]) --> lSuccess
```

Arguments

<xIdentity> is a unique value guaranteed by the structure of the data file to reference a specific item in a data source (database). In a (.dbf) <xIdentity> is the record number. In other data formats, <xIdentity> is the unique primary key value.

Returns

DBRLOCK() returns lSuccess, a logical data type that is true (.T.) if successful, false (.F.) if unsuccessful.

Description

DBRLOCK() is a database function that locks the record identified by the value <xIdentity>. In Xbase, <xIdentity> is the record number.

If you do not specify <xIdentity>, all record locks are released and the current record is locked. If you specify <xIdentity>, DBRLOCK() attempts to lock it and, if successful, adds it to the locked record list.

Examples

. This example shows two different methods for locking multiple records:

```
FUNCTION dbRLockRange( nLo, nHi )

   LOCAL nRec
   FOR nRec := nLo TO nHi
      IF ! DBRLOCK( nRec )
         DBRUNLOCK()         // Failed - unlock everything
      ENDIF
   NEXT
   RETURN DBRLOCKLIST()      // Return array of actual locks

FUNCTION dbRLockArray( aList )

   LOCAL nElement, nLen, lRet
   lRet := .T.
   nLen := LEN( aList )
   FOR nElement := 1 TO nLen
      IF ! DBRLOCK( aList[ nElement ] )
         DBRUNLOCK()         // Failed - unlock everything
         lRet := .F.
      ENDIF
   NEXT
   RETURN DBRLOCKLIST()
```

Status: New

See also: DBUNLOCK(), DBUNLOCKALL(), FLOCK(), RLOCK(), UNLOCK

See Also: DBUNLOCK() DBUNLOCKALL() FLOCK() RLOCK() UNLOCK

DBRLOCKLIST() function
Return an array of the current Lock List

Syntax

```
DBRLOCKLIST() --> aRecordLocks
```

Returns

Returns an array of the locked records in the current or aliased work area.

Description

DBRLOCKLIST() is a database function that returns a one-dimensional array that contains the identities of record locks active in the selected work area.

Examples

```
PROCEDURE PrintCurLocks()

   LOCAL aList
   LOCAL nSize
   LOCAL nCount

   aList := DBRLOCKLIST()
   nSize := LEN( aList )

   ? "Currently locked records: "
   FOR nCount := 1 TO nSize
      ?? aList[ nCount ]
      ?? SPACE( 1 )
   NEXT
   ?

   RETURN
```

Status: New

See also: RLOCK(), UNLOCK, DBRLOCK(), DBRUNLOCK()

See Also: RLOCK() UNLOCK DBRLOCK() DBRUNLOCK()

DBRUNLOCK() function
Release all or specified record locks

Syntax

```
DBRUNLOCK([<xIdentity>]) --> NIL
```

Arguments

<xIdentity> is a unique value guaranteed by the structure of the data file to reference a specific item in a data source (database). In a (.dbf) <xIdentity> is the record number. In other data formats, <xIdentity> is the unique primary key value.

Returns

DBRUNLOCK() always returns NIL.

Description

DBRUNLOCK() is a database function that releases the lock on <xIdentity> and removes it from the Lock List. If <xIdentity> is not specified, all record locks are released.

Examples

```
PROCEDURE dbRUnlockRange( nLo, nHi )

   LOCAL nCounter

   // Unlock the records in the range from nLo to nHi
   FOR nCounter := nLo TO nHi
      DBRUNLOCK( nCounter )
   NEXT

   RETURN
```

Status: New

See also: RLOCK(), DBRLOCK(), DBRLOCKLIST()

See Also: RLOCK() DBRLOCK() DBRLOCKLIST()

DBSETINDEX() function*
Empty Orders from an Order Bag into the Order List

Syntax

DBSETINDEX(**<cOrderBagName>**) --> NIL

Arguments

<cOrderBagName> is the name of a disk file containing one or more Orders. You may specify <cOrderBagName> as the filename with or without the pathname or extension. If you do not include the extension as part of <cOrderBagName>, CA-Clipper uses the default extension of the current RDD.

Returns

DBSETINDEX() always returns NIL.

Description

DBSETINDEX() is a database function that adds the contents of an Order Bag into the Order List of the current work area. Any Orders already associated with the work area continue to be active. If the newly opened Order Bag is the only Order associated with the work area, it becomes the controlling Order; otherwise, the controlling Order remains unchanged. If the Order Bag contains more than one Order, and there are no other Orders associated with the work area, the first Order in the new Order Bag becomes the controlling Order.

DBSETINDEX() is a compatibility command and therefore not recommended. It is superseded by the ORDLISTADD() function.

Examples

```
USE Customer NEW
DBSETINDEX( "Cust01" )        // Open the index Cust01
                              // in the current work area
DBSETINDEX( "Cust02" )        // Open the index Cust02
                              // leaving Cust01 open
```

Status: Enhanced

See also: ORDLISTADD()

See Also: ORDLISTADD()

DELETE TAG command
Delete a Tag

Syntax

```
DELETE TAG <cOrderName> [IN <xcOrderBagName>]
   [, <cOrderName> [IN xcOrderBagName] list>]
```

Arguments

<cOrderName> is a character string that represents the Order Name.

<xcOrderBagName> is the name of a disk file containing one or more Orders. You may specify <xcOrderBagName> as the filename with or without the pathname or appropriate extension. If you do not include the extension as part of <xcOrderBagName> CA-Clipper uses the default extension of the current RDD.

<cOrderName ...list> is an optional list of Order and Order Bag Name pairs, separated by commas. Any reference to <cOrderName> that results in either a null string ("") or spaces is ignored. You can specify each Order as a literal expression or as a character expression enclosed in parentheses. If you specify no extension for the Order Bag Name, the current database driver supplies a default extension.

Description

This command removes an Order from an Order Bag in the current or specified work area. If you do not specify an <xcOrderBagName>, all Orders Bags are searched in the current or specified work area. The first occurrence of <cOrderName> is deleted. A runtime recoverable error is raised if the Order is not found.

If <cOrderName> is the active Order, the database in the current or specified work area reverts to its identity Order (natural or entry) order and SET FILTER scoping.

A runtime error is raised if <xcOrderBagName> does not exist or if it exists but does not contain <cOrderName>.

The active RDD determines the Order capacity of an Order Bag. The default DBFNTX and the DBFNDX drivers only support single-Order Bags, while other RDDs may support multiple-Order Bags (e.g., the DBFCDX and DBFMDX drivers).

Examples

```
USE Customer VIA "DBFCDX" NEW
SET INDEX TO Customer

// Delete the Orders (Tags) Cust01 and Cust02 that
// exist in the index file Customer
DELETE TAG Cust01 IN Customer
DELETE TAG Cust02 IN Customer

// or
// DELETE TAG Cust01 IN Customer, Cust02 IN Customer
```

Status: New

See also: ORDDESTROY()

See Also: ORDDESTROY()

GO command
Move the pointer to the specified identity

Syntax

GO[TO] **<xIdentity>**

Arguments

<xIdentity> a unique value guaranteed by the structure of the data file to reference a specific item in a data source (database). In a (.dbf), identity is the record number. In other data formats, identity is the unique primary key value.

Description

GO[TO] is a database command that positions the record pointer in the current work area at the specified identity. In an Xbase data structure, this identity is the record number because every record, even an empty record, has a record number. In data structures of different design, identity may be defined as something other than record number.

Examples

. This example saves the current record number, searches for a key, then restores the record pointer to the saved position:

```
FUNCTION KeyExists( xKeyExpr )

   LOCAL nSavRecord := RECNO()      // Save the current record
                                    // pointer position
   LOCAL lFound

   SEEK xKeyExpr
   IF ( lFound := FOUND() )
      .
      .  < statements >
      .
   ENDIF

   GOTO nSavRecord      // Restore the record pointer position

   RETURN ( lFound )
```

Status: Enhanced

See also: DBGOTO(), LASTREC(), RECNO(), SET DELETED, SET FILTER, SET RELATION, SKIP

See Also: DBGOTO() LASTREC() RECNO() SET DELETED SET FILTER

INDEX command
Create an index file

Syntax

```
INDEX ON <expKey> [TAG <cOrderName>] TO
   <xcOrderBagName> [FOR <lCondition>] [ALL]
   [WHILE <lCondition>] [NEXT <nNumber>]
   [RECORD <nRecord>] [REST]
   [EVAL <bBlock> [EVERY <nInterval>]
   [UNIQUE] [ASCENDING|DESCENDING]

OR

INDEX ON <expKey> TAG <cOrderName> [TO
   <xcOrderBagName>] [FOR <lCondition>] [ALL]
   [WHILE <lCondition>] [NEXT <nNumber>]
   [RECORD <nRecord>] [REST]
   [EVAL <bBlock> [EVERY <nInterval>]
   [UNIQUE] [ASCENDING|DESCENDING]
```

Arguments

<expKey> is an expression that returns the key value to place in the index for each record in the current work area. <expKey> can be character, date, logical, or numeric type. The maximum length of the index key expression is determined by the driver.

TAG <cOrderName> is the name of the Order to be created. In (.ntx) and (.ndx) files the filename, without extension or pathname, is the Order Name if none is specified. <cOrderName> can be any CA-Clipper expression that evaluates to a string constant.

TO <xcOrderBagName> is the name of a disk file containing one or more Orders. The active RDD determines the Order capacity of an Order Bag. The default DBFNTX driver only supports single-Order Bags, while other RDDs may support multiple-Order Bags (e.g., the DBFCDX and DBFMDX drivers). You may specify <xcOrderBagName> as the filename with or without pathname or extension. If an extension is not provided as part of <xcOrderBagName> CA-Clipper will use the default extension of the current RDD.

Both the TAG and the TO clauses are optional, but you must use at least one of them.

FOR <lCondition> specifies the conditional set of records on which to create the Order. Only those records that meet the condition are included in the resulting Order. <lCondition> is an expression that may be no longer than 250 characters under the DBFNTX and DBFNDX drivers. The maximum value for these expressions is determined by the RDD. The FOR condition is stored as part of the Order Bag and used when updating or recreating the index using the REINDEX command. Duplicate key values are not added to the Order Bag.

Drivers that do not support the FOR condition will produce an "unsupported" error.

The FOR clause provides the only scoping that is maintained for all database changes. All other scope conditions create Orders that do not reflect database updates.

ALL specifies all Orders in the current or specified work area. ALL is the default scope of INDEX ON.

WHILE <lCondition> specifies another condition that must be met by each record as it is processed. As soon as a record is encountered that causes the condition to fail, the INDEX command terminates. If a WHILE

clause is specified, the data is processed in the controlling order. The WHILE condition is transient (i.e., it is not stored in the file and not used for index updates and REINDEXing purposes). The WHILE clause creates temporary Orders, but these Orders are not updated.

Drivers that do not support the WHILE condition will produce an "unsupported" error.

Using the WHILE clause is more efficient and faster than using the FOR clause. The WHILE clause only processes data for which <lCondition> is true from the current position. The FOR clause, however, processes all data in the data source.

NEXT <nNumber> specifies the portion of the database to process. If you specify NEXT, the database is processed in the controlling order for the <nNumber> number of identities. The scope is transient (i.e., is not stored in the Order and not used for REINDEXing purposes).

RECORD <nRecord> specifies the processing of the specified record.

REST specifies the processing of all records from the current position of the record pointer to the end of file (EOF).

EVAL <bBlock> evaluates a code block every <nInterval>, where <nInterval> is a value specified by the EVERY clause. The default value is 1. This is useful in producing a status bar or odometer that monitors the indexing progress. The return value of <bBlock> must be a logical data type. If <bBlock> returns false (.F.), indexing halts.

EVERY <nInterval> is a clause containing a numeric expression that modifies the number of times <bBlock> is EVALuated. The EVERY option of the EVAL clause offers a performance enhancement by evaluating the condition for every nth record instead of for every record ordered. The EVERY keyword is ignored if you specify no EVAL condition.

UNIQUE specifies that the key value of each record inserted into the Order be unique. Duplicate key values are not added to the Order.

ASCENDING specifies that the keyed pairs be sorted in increasing order of value. If neither ASCENDING nor DESCENDING is specified, ASCENDING is assumed. Although not stored as an explicit part of the file, ASCENDING is an implicit file attribute that is understood by the REINDEX command.

Drivers that do not support the ASCENDING condition will produce an "unsupported" error.

DESCENDING specifies that the keyed pairs be sorted in decreasing order of value. Using this keyword is the same as specifying the DESCEND() function within <expKey>, but without the performance penalty during Order updates. If you create a DESCENDING index, you won't need to use the DESCEND() function during a SEEK. DESCENDING is an attribute of the file, where it is stored and used for REINDEXing purposes.

Drivers that do not support the DESCENDING condition will produce an "unsupported" error.

Description

The INDEX ON command adds a set of keyed pairs, ordered by <expKey> to a file specified by <xcOrderBagName> using the database open in the current work area.

In RDDs that support production or structural indexes (e.g., DBFCDX, DBFMDX), if you specify a Tag but do not specify an Order Bag, the Tag is created and added to the Order Bag. If no production or structural index exists, it will be created and the Tag will be added to it. When using RDDs that support multiple Order Bags, you must explicitly SET

ORDER (or ORDSETFOCUS()) to the desired controlling Order. If you do not specify a controlling Order, the data file will be viewed in natural Order.

If <xcOrderBagName> does not exist, it is created in accordance with the RDD in the current or specified work area.

If <xcOrderBagName> exists and the RDD specifies that Order Bags can only contain a single Order, <xcOrderBagName> is erased and the new Order is added to the Order Bag and to the Order List in the current or specified work area.

If <xcOrderBagName> exists and the RDD specifies that Order Bags can contain multiple Tags, <cOrderName> is created if it does not already exist, otherwise <cOrderName> is replaced in <xcOrderBagName> and the Order is added to the Order List in the current or specified work area.

ASCENDING or DESCENDING specifies the sequence of keyed pairs in the Order. If neither clause is specified, the default is ASCENDING.

If you specify the UNIQUE clause, the resulting Order will contain only unique records. Some RDDs may do this by only including record references to a key value once. Others may produce a runtime recoverable error as a nonunique key insertion is attempted.

The EVAL clause lets you specify a code block to be evaluated as each record is placed in the Order. The EVERY clause lets you modify how often <bBlock> is called. Instead of evaluation as each record is placed in the Order, evaluation only occurs as every <nInterval> records are placed in the Order.

The INDEX ON command accepts certain clauses that let the user create conditional and partial Orders. Some Orders are intended to be maintained across the application, others are considered "temporary" Orders.

The FOR clause provides the only Order scoping that is permanent and can be maintained across the life of the application. The string passed as the FOR condition is stored within the Order for later use in maintaining the Order. Though only accessing part of a database, Orders created using this clause exist as long as the database is active. The FOR clause lets you create maintainable scoped Orders.

The WHILE, NEXT, REST and RECORD clauses process data from the current position of the database cursor in the default or specified work area. If you specify these clauses, the Order List remains open and the active Order is used to organize the database while it is being created. These clauses let you create temporary (non-maintainable) Orders. Orders created using these clauses contain records in which <lCondition> is true (.T.) at the location of the record pointer.

Examples

. The following example creates a simple order (index) based on one field (Acct):

```
USE Customer NEW
INDEX ON Customer->Acct TO CuAcct
```

. This example creates a conditional Order (index) based on a FOR clause. This index will contain only records whose field TransDate contains a date greater than or equal to January 1, 1992:

```
USE Invoice NEW
INDEX ON Invoice->TransDate   ;
   TO InDate                  ;
   FOR ( Invoice->TransDate >= CTOD( "01/01/92" ) )
```

. This example creates an order in a multiple-Order Bag (i.e. a tag in an index that can support multiple tags in an index file):

```
USE Customer NEW
INDEX ON Customer->Acct TAG CuAcct TO Customer
```

. The following example creates an order that calls a routine "MyMeter" during its creation:

```
#define MTR_INCREMENT   10

USE Customer NEW
INDEX ON Customer->Acct TO CuAcct EVAL ;
      {|| MyMeter() } EVERY MTR_INCREMENT

FUNCTION MyMeter()

   STATIC nRecsDone := 0

   nRecsDone := += MTR_INCREMENT
   ? ( nRecsDone/LASTREC() ) * 100

   RETURN ( .T. )
```

Status: Enhanced

See also: CLOSE, DESCEND(), DTOS(), INDEXKEY(), INDEXORD(), ORDCREATE(), REINDEX, SEEK, SET INDEX, SET ORDER, SORT, SOUNDEX(), USE

See Also: CLOSE DESCEND() DTOS() INDEXKEY() INDEXORD()

ORDBAGEXT() function
Return the default Order Bag RDD extension

Syntax

```
ORDBAGEXT() --> cBagExt
```

Returns

ORDBAGEXT() returns a character expression.

Description

ORDBAGEXT() is an Order management function that returns a character expression that is the default Order Bag extension of the current or aliased work area. cBagExt is determined by the RDD active in the current work area.

ORDBAGEXT() supersedes the INDEXEXT() and is not recommended.

Examples

```
USE sample VIA "DBFNTX"
? ORDBAGEXT()      //  Returns .ntx
```

Status: New

See also: INDEXEXT(), ORDBAGNAME()

See Also: INDEXEXT() ORDBAGNAME()

ORDBAGNAME() function
Return the Order Bag name of a specific Order

Syntax

```
ORDBAGNAME(<nOrder> | <cOrderName>) --> cOrderBagName
```

Arguments

<nOrder> is an integer that identifies the position in the Order List of the target Order whose Order Bag name is sought.

<cOrderName> is a character string that represents the name of the target Order whose Order Bag name is sought.

Returns

ORDBAGNAME() returns a character string, the Order Bag name of the specific Order.

Description

ORDBAGNAME() is an Order management function that lets you access the name of the Order Bag in which <cOrderName> resides. You may identify the Order as a character string or with an integer that represents its position in the Order List. In case of duplicate names, ORDBAGNAME() only recognizes the first matching name.

Examples

. The following example uses ORDBAGNAME() with the default DBFNTX driver:

```
USE Customer VIA "DBFNTX" NEW
SET INDEX TO CuAcct, CuName, CuZip
ORDBAGNAME( 2 )               // Returns: CuName
ORDBAGNAME( 1 )               // Returns: CuAcct
ORDBAGNAME( 3 )               // Returns: CuZip
```

. In this example, Customer.cdx contains three orders named CuAcct, CuName, CuZip:

```
USE Customer VIA "DBFCDX" NEW
SET INDEX TO Customer
ORDBAGNAME( "CuAcct" )        // Returns: Customer
ORDBAGNAME( "CuName" )        // Returns: Customer
ORDBAGNAME( "CuZip" )         // Returns: Customer
```

Status: New

See also: ORDBAGEXT()

See Also: ORDBAGEXT()

ORDCREATE() function
Create an Order in an Order Bag

Syntax

```
ORDCREATE(<cOrderBagName>,[<cOrderName>],    <cExpKey>,
   [<bExpKey>], [<lUnique>]) --> NIL
```

Arguments

<cOrderBagName> is the name of a disk file containing one or more Orders. You may specify <cOrderBagName> as the filename with or without the pathname or extension. If you do not include the extension as part of <cOrderBagName> CA-Clipper uses the default extension of the current RDD.

<cOrderName> is the name of the Order to be created.

Note: Although both <cOrderBagName> and <cOrderName> are both optional, at least one of them must be specified.

<cExpKey> is an expression that returns the key value to place in the Order for each record in the current work area. <cExpKey> can represent a character, date, logical, or numeric data type. The maximum length of the index key expression is determined by the database driver.

<bExpKey> is a code block that evaluates to a key value that is placed in the Order for each record in the current work area. If you do not supply <bExpKey>, it is macro-compiled from <cExpKey>.

<lUnique> is an optional logical value that specifies whether a unique Order is to be created. If <lUnique> is omitted, the current global _SET_UNIQUE setting is used.

Returns

ORDCREATE() always returns NIL.

Description

ORDCREATE() is an Order management function that creates an Order in the current work area. It works like DBCREATEINDEX() except that it lets you create Orders in RDDs that recognize multiple-Order Bags. ORDCREATE() supersedes the DBCREATEINDEX() function because of this capability, and is the preferred function.

The active RDD determines the Order capacity of an Order Bag. The default DBFNTX and the DBFNDX drivers only support single-Order Bags, while other RDDs may support multiple-Order Bags (e.g., the DBFCDX and DBFMDX drivers).

In RDDs that support production or structural indexes (e.g., DBFCDX, DBPX), if you specify a Tag but do not specify an Order Bag, the Tag is created and added to the index. If no production or structural index exists, it will be created and the Tag will be added to it. When using RDDs that support multiple Order Bags, you must explicitly SET ORDER (or ORDSETFOCUS()) to the desired controlling Order. If you do not specify a controlling Order, the data file will be viewed in natural Order.

If <cOrderBagName> does not exist, it is created in accordance with the RDD in the current or specified work area.

If <cOrderBagName> exists and the RDD specifies that Order Bags can only contain a single Order, <cOrderBagName> is erased and the new Order is added to the Order List in the current or specified work area.

If <cOrderBagName> exists and the RDD specifies that Order Bags can

contain multiple Tags, <cOrderName> is created if it does not already exist, otherwise <cOrderName> is replaced in <cOrderBagName> and the Order is added to the Order List in the current or specified work area.

Examples

. The following example demonstrates ORDCREATE() with the DBFNDX driver:

```
USE Customer VIA "DBFNDX" NEW
ORDCREATE( "CuAcct",, "Customer->Acct" )
```

. The following example demonstrates ORDCREATE() with the default DBFNTX driver:

```
USE Customer VIA "DBFNTX" NEW
ORDCREATE( "CuAcct", "CuAcct", "Customer->Acct", ;
      {|| Customer->Acct } )
```

. The following example demonstrates ORDCREATE() with the FoxPro driver, DBFCDX:

```
USE Customer VIA "DBFCDX" NEW
ORDCREATE( "Customer", "CuAcct", "Customer->Acct" )
```

. This example creates the Order "CuAcct" and adds it to the production index (Order Bag) "Customer". The production index , will be created if it doesn't exist:

```
USE Customer VIA "DBFMDX" NEW
ORDCREATE( , "CuAcct", "Customer->Acct" )
```

Status: New

See also: DBCREATEIND(), INDEX, SET UNIQUE*

See Also: DBCREATEIND() INDEX SET UNIQUE*

ORDDESTROY() function
Remove a specified Order from an Order Bag

Syntax

```
ORDDESTROY(<cOrderName> [, <cOrderBagName> ]) --> NIL
```

Arguments

<cOrderName> is the name of the Order to be removed from the current or specified work area.

<cOrderBagName> is the name of a disk file containing one or more Orders. You may specify <cOrderBagName> as the filename with or without the pathname or appropriate extension. If you do not include the extension as part of <cOrderBagName> CA-Clipper uses the default extension of the current RDD.

Returns

ORDDESTROY() always returns NIL.

Description

ORDDESTROY() is an Order management function that removes a specified Order from multiple-Order Bags.

The active RDD determines the Order capacity of an Order Bag. The default DBFNTX and the DBFNDX drivers only support single-Order Bags, while other RDDs may support multiple-Order Bags (e.g., the DBFCDX and DBPX drivers).

Note: RDD suppliers may define specific behaviors for this command.

Warning! ORDDESTROY() is not supported for DBFNDX and DBFNTX.

Examples

. This example demonstrates ORDDESTROY() with the FoxPro driver, DBFCDX:

```
USE Customer VIA "DBFCDX" NEW
SET INDEX TO Customer, CustTemp
ORDDESTROY( "CuAcct", "Customer" )
```

Status: New

See also: DELETE TAG, ORDCREATE()

See Also: DELETE TAG ORDCREATE()

ORDFOR() function
Return the FOR expression of an Order

Syntax

```
ORDFOR(<cOrderName> | <nOrder>
   [, <cOrderBagName>]) --> cForExp
```

Arguments

<cOrderName> is the name of the target Order, whose cForExp is sought.

<nOrder> is an integer that identifies the position in the Order List of the target Order whose cForExp is sought.

<cOrderBagName> is the name of an Order Bag containing one or more Orders. You may specify <cOrderBagName> as the filename with or without the pathname or appropriate extension. If you do not include the extension as part of <cOrderBagName> CA-Clipper uses the default extension of the current RDD.

Returns

ORDFOR() returns a character expression, cForExp, that represents the FOR condition of the specified Order. If the Order was not created using the FOR clause the return value will be an empty string (""). If the database driver does not support the FOR condition, it may either return an empty string ("") or raise an "unsupported function" error, depending on the driver.

Description

ORDFOR() is an Order management function that returns the character string, cForExp, that represents the logical FOR condition of the Order, <cOrderName> or <nOrder>.

Examples

. This example retrieves the FOR condition from an Order:

```
USE Customer NEW
INDEX ON  Customer->Acct ;
   TO  Customer          ;
   FOR Customer->Acct > "AZZZZZ"

ORDFOR( "Customer" )      // Returns: Customer->Acct > "AZZZZZ"
```

Status: New

See also: INDEX, ORDKEY(), ORDCREATE(), ORDNAME(), ORDNUMBER()

See Also: INDEX ORDKEY() ORDCREATE() ORDNAME() ORDNUMBER()

ORDKEY() function
Return the key expression of an Order

Syntax

```
ORDKEY(<cOrderName> | <nOrder>
   [, <cOrderBagName>]) --> cExpKey
```

Arguments

<cOrderName> is the name of an Order, a logical ordering of a database.

<nOrder> is an integer that identifies the position in the Order List of the target Order whose cExpKey is sought.

<cOrderBagName> is the name of a disk file containing one or more Orders. You may specify <cOrderBagName> as the filename with or without the pathname or appropriate extension. If you do not include the extension as part of <cOrderBagName> CA-Clipper uses the default extension of the current RDD.

Returns

Returns a character string, cExpKey.

Description

ORDKEY() is an Order management function that returns a character expression, cExpKey, that represents the key expression of the specified Order.

You may specify the Order by name or with a number that represents its position in the Order List. Using the Order name is the preferred method.

The active RDD determines the Order capacity of an Order Bag. The default DBFNTX and the DBFNDX drivers only support single-Order Bags, while other RDDs may support multiple-Order Bags (e.g., the DBFCDX and DBFMDX drivers).

Examples

. This example retrieves the index expression from an Order:

```
USE Customer NEW
INDEX ON  Customer->Acct  ;
   TO  Customer           ;
   FOR Customer->Acct > "AZZZZZ"

ORDKEY( "Customer" )      // Returns: Customer->Acct
```

Status: New

See also: ORDFOR(), ORDNAME(), ORDNUMBER()

See Also: ORDFOR() ORDNAME() ORDNUMBER()

ORDLISTADD() function
Add Orders to the Order List

Syntax

```
ORDLISTADD(<cOrderBagName>
   [, <cOrderName>]) --> NIL
```

Arguments

<cOrderBagName> is the name of a disk file containing one or more Orders. You may specify <cOrderBagName> as the filename with or without the pathname or appropriate extension. If you do not include the extension as part of <cOrderBagName> CA-Clipper uses the default extension of the current RDD.

<cOrderName> the name of the specific Order from the Order Bag to be added to the Order List of the current work area. If you do not specify <cOrderName>, all orders in the Order Bag are added to the Order List of the current work area.

Returns

ORDLISTADD() always returns NIL.

Description

ORDLISTADD() is an Order management function that adds the contents of an Order Bag , or a single Order in an Order Bag, to the Order List. This function lets you extend the Order List without issuing a SET INDEX command that, first, clears all the active Orders from the Order List.

Any Orders already associated with the work area continue to be active. If the newly opened Order Bag contains the only Order associated with the work area, it becomes the controlling Order; otherwise, the controlling Order remains unchanged.

After the new Orders are opened, the work area is positioned to the first logical record in the controlling Order.

ORDLISTADD() is similar to the SET INDEX command or the INDEX clause of the USE command, except that it does not clear the Order List prior to adding the new order(s).

ORDLISTADD() supersedes the DBSETINDEX() function.

The active RDD determines the Order capacity of an Order Bag. The default DBFNTX and the DBFNDX drivers only support single-Order Bags, while other RDDs may support multiple-Order Bags (e.g., the DBFCDX and DBPX drivers). When using RDDs that support multiple Order Bags, you must explicitly SET ORDER (or ORDSETFOCUS()) to the desired controlling Order. If you do not specify a controlling Order, the data file will be viewed in natural Order.

Examples

. In this example Customer.cdx contains three orders, CuAcct, CuName, and CuZip. ORDLISTADD() opens Customer.cdx but only uses the order named CuAcct:

```
USE Customer VIA "DBFCDX" NEW
ORDLISTADD( "Customer", "CuAcct" )
```

Status: New

See also: DBSETINDEX(), INDEX, SET INDEX, USE

See Also: DBSETINDEX()* INDEX SET INDEX USE

ORDLISTCLEAR() function
Clear the current Order List

Syntax

```
ORDLISTCLEAR() --> NIL
```

Returns

ORDLISTCLEAR() always returns NIL.

Description

ORDLISTCLEAR() is an Order management function that removes all Orders from the Order List for the current or aliased work area. When you are done, the Order List is empty.

This function supersedes the function DBCLEARINDEX().

Examples

```
USE Sales NEW
SET INDEX TO SaRegion, SaRep, SaCode
.
. < statements >
.
ORDLISTCLEAR()      // Closes all the current indexes
```

Status: New

See also: SET INDEX, DBCLEARINDEX()

See Also: SET INDEX DBCLEARIND()

ORDLISTREBUILD() function

Rebuild all Orders in the Order List of the current work area

---

Syntax

```
ORDLISTREBUILD() --> NIL
```

Returns

ORDLISTREBUILD() always returns NIL.

Description

ORDLISTREBUILD() is an Order management function that rebuilds all the orders in the current or aliased Order List.

To only rebuild a single Order use the function ORDCREATE().

Unlike ORDCREATE(), this function rebuilds all Orders in the Order List. It is equivalent to REINDEX.

Examples

```
USE Customer NEW
SET INDEX TO CuAcct, CuName, CuZip
ORDLISTREBUILD()     // Causes CuAcct, CuName, CuZip to
                     // be rebuilt
```

Status: New

See also: REINDEX, INDEX, ORDCREATE()

See Also: REINDEX INDEX ORDCREATE()

ORDNAME() function
Return the name of an Order in the Order List

Syntax

```
ORDNAME(<nOrder>[,<cOrderBagName>])
   --> cOrderName
```

Arguments

<nOrder> is an integer that identifies the position in the Order List of the target Order whose database name is sought.

<cOrderBagName> is the name of a disk file containing one or more Orders. You may specify <cOrderBagName> as the filename with or without the pathname or appropriate extension. If you do not include the extension as part of <xcOrderBagName> CA-Clipper uses the default extension of the current RDD.

Returns

ORDNAME() returns the name of the specified Order in the current Order List or the specified Order Bag if opened in the Current Order list.

Description

ORDNAME() is an Order management function that returns the name of the specified Order in the current Order List.

If <cOrderBagName> is an Order Bag that has been emptied into the current Order List, only those Orders in the Order List that correspond to <cOrderBagName> Order Bag are searched.

The active RDD determines the Order capacity of an Order Bag. The default DBFNTX and the DBFNDX drivers only support single-Order Bags, while other RDDs may support multiple-Order Bags (e.g., the DBFCDX and DBPX drivers).

Examples

. This example retrieves the name of an Order using its position in the order list:

```
USE Customer NEW
SET INDEX TO CuAcct, CuName, CuZip
ORDNAME( 2 )                        // Returns: CuName
```

. This example retrieves the name of an Order given its position within a specific Order Bag in the Order List:

```
USE Customer NEW
SET INDEX TO Temp, Customer
// Assume Customer contains CuAcct, CuName, CuZip
ORDNAME( 2, "Customer" )            // Returns: CuName
```

Status: New

See also: ORDFOR(), ORDKEY(), ORDNUMBER()

See Also: ORDFOR() ORDKEY() ORDNUMBER()

ORDNUMBER() function
Return the position of an Order in the current Order List

Syntax

```
ORDNUMBER(<cOrderName>
   [, <cOrderBagName>]) --> nOrderNo
```

Arguments

<cOrderName> the name of the specific Order whose position in the Order List is sought.

<cOrderBagName> is the name of a disk file containing one or more Orders. You may specify <cOrderBagName> as the filename with or without the pathname or appropriate extension. If you do not include the extension as part of <cOrderBagName> CA-Clipper uses the default extension of the current RDD.

Returns

Returns nOrderNo, an integer that represents the position of the specified Order in the Order List.

Description

ORDNUMBER() is an Order management function that lets you determine the position in the current Order List of the specified Order. ORDNUMBER() searches the Order List in the current work area and returns the position of the first Order that matches <cOrderName>. If <cOrderBagName> is the name of an Order Bag newly emptied into the current Order List, only those orders in the Order List that have been emptied from <cOrderBagName> are searched.

If <cOrderName> is not found ORDNUMBER() raises a recoverable runtime error.

The active RDD determines the Order capacity of an Order Bag. The default DBFNTX driver only supports single-Order Bags, while other RDDs may support multiple-Order Bags (e.g., the DBFCDX and DBPX drivers).

Examples

```
USE Customer VIA "DBFNTX" NEW
SET INDEX TO CuAcct, CuName, CuZip
ORDNUMBER( "CuName" )            // Returns: 2
```

Status: New

See also: INDEXORD()

See Also: INDEXORD()

ORDSETFOCUS() function
Set focus to an Order in an Order List

Syntax

```
ORDSETFOCUS([<cOrderName> | <nOrder>]
   [,<cOrderBagName>]) --> cPrevOrderNameInFocus
```

Arguments

<cOrderName> is the name of the selected Order, a logical ordering of a database. ORDSETFOCUS() ignores any invalid values of <cOrderName>.

<nOrder> is a number representing the position in the Order List of the selected Order.

<cOrderBagName> is the name of a disk file containing one or more Orders. You may specify <cOrderBagName> as the filename with or without the pathname or appropriate extension. If you do not include the extension as part of <cOrderBagName> CA-Clipper uses the default extension of the current RDD.

Returns

ORDSETFOCUS() returns the Order Name of the previous controlling Order.

Description

ORDSETFOCUS() is an Order management function that returns the Order Name of the previous controlling Order and optionally sets the focus to an new Order.

If you do not specify <cOrderName> or <nOrder>, the name of the currently controlling order is returned and the controlling order remains unchanged.

All Orders in an Order List are properly updated no matter what <cOrderName> is the controlling Order. After a change of controlling Orders, the record pointer still points to the same record.

The active RDD determines the Order capacity of an Order Bag. The default DBFNTX driver only supports single-Order Bags, while other RDDs may support multiple-Order Bags (e.g., the DBFCDX and DBPX drivers).

ORDSETFOCUS() supersedes INDEXORD().

Examples

```
USE Customer VIA "DBFNTX" NEW
SET INDEX TO CuAcct, CuName, CuZip
? ORDSETFOCUS( "CuName" )        // Displays: "CuAcct"
? ORDSETFOCUS()                  // Displays: "CuName"
```

Status: New

See also: SET INDEX, SET ORDER

See Also: SET INDEX SET ORDER

RDDLIST() function
Return an array of the available Replaceable Database Drivers

---

Syntax

```
RDDLIST([<nRDDType>]) --> aRDDList
```

Arguments

<nRDDType> is an integer that represents the type of the RDD you wish to list. The constants RDT_FULL and RDT_TRANSFER represent the two types of RDDs currently available.

RDDType Summary

Constant	Value	Meaning
RDT_FULL	1	Full RDD implementation
RDT_TRANSFER	2	Import/Export only driver

RDT_FULL identifies full-featured RDDs that have all the capabilities associated with an RDD.

RDT_TRANSFER identifies RDDs of limited capability. They can only transfer records between files. You cannot use these limited RDD drivers to open a file in a work area. The SDF and DELIM drivers are examples of this type of RDD. They are only used in the implementation of APPEND FROM and COPY TO with SDF or DELIMITED files.

Returns

RDDLIST() returns a one-dimensional array of the RDD names registered with the application as <nRDDType>.

Description

RDDLIST() is an RDD function that returns a one-dimensional array that lists the available RDDs.

If you do not supply <nRDDType>, all available RDDs, regardless of type, are returned.

Examples

. In this example RDDLIST() returns an array containing the character strings, "DBF", "SDF", "DELIM", "DBFCDX", and "DBFNTX":

```
REQUEST DBFCDX

.
. < statements >
.

aRDDs := RDDLIST()

      // Returns {"DBF", SDF", "DELIM", "DBFCDX", "DBFNTX" }
```

. In this example, RDDLIST() returns an array containing the character strings, "SDF" and "DELIM":

```
#include "rddsys.ch"
.
. < statements >
.
aImpExp := RDDLIST( RDT TRANSFER )
```

Files: RDDSYS.CH

Status: New

See also: DBSETDRIVER(), RDDNAME(), RDDSETDEFAULT()

See Also: DBSETDRIVER() RDDNAME() RDDSETDEFAULT()

RDDNAME() function

Return the name of the RDD active in the current or specified work area

---

Syntax

```
RDDNAME() --> cRDDName
```

Returns

Returns a character string, cRDDName, the registered name of the active RDD in the current or specified work area.

Description

RDDNAME() is an RDD function that returns a character string, cRDDName, the name of the active RDD in the current or specified work area.

You can specify a work area other than the currently active work area by aliasing the function.

Examples

```
USE Customer VIA "DBFNTX" NEW
USE Sales    VIA "DBFCDX" NEW

? RDDNAME()                          // Returns: DBFCDX
? Customer->( RDDNAME() )            // Returns: DBFNTX
? Sales->( RDDNAME() )               // Returns: DBFCDX
```

Status: New

See also: RDDLIST()

See Also: RDDLIST()

RDDSETDEFAULT() function
Set or return the default RDD for the application

Syntax

```
RDDSETDEFAULT([<cNewDefaultRDD>])
   --> cPreviousDefaultRDD
```

Arguments

<cNewDefaultRDD> is a character string, the name of the RDD that is to be made the new default RDD in the application.

Returns

RDDSETDEFAULT() returns a character string, cPreviousDefaultRDD, the name of the previous default driver. The default driver is the driver that CA-Clipper uses if you do not explicitly specify an RDD with the VIA clause of the USE command.

Description

RDDSETDEFAULT() is an RDD function that sets or returns the name of the previous default RDD driver and, optionally, sets the current driver to the new RDD driver specified by cNewDefaultRDD. If <cNewDefaultDriver> is not specified, the current default driver name is returned and continues to be the current default driver.

This function replaces the DBSETDRIVER() function.

Examples

```
// If the default driver is not DBFNTX, make it the default

IF ( RDDSETDEFAULT() != "DBFNTX" )
   cOldRdd := RDDSETDEFAULT( "DBFNTX" )
ENDIF
```

Status: New

See also: DBSETDRIVER()

See Also: DBSETDRIVER()

RECNO() function
Return the identity at the position of the record pointer

Syntax

RECNO() --> **Identity**

Returns

RECNO() returns the identity found at the position of the record pointer.

Description

RECNO() is a database function that returns the identity found at the current position of the record pointer. Identity is a unique value guaranteed by the structure of the data file to reference a specific record of data file. The data file need not be a traditional Xbase file. Therefore, unlike earlier versions of CA-Clipper, the value returned need not be a numeric data type.

Under all RDDs, RECNO() returns the value at the position of the record pointer; the data type and other characteristics of this value are determined by the content of the accessed data and the RDD active in the current work area. In an Xbase database this value is the record number.

Examples

```
USE Sales VIA "DBFNTX"
.
. < statements >
.
DBGOTOP()
RECNO()            // Returns 1
```

Status: Enhanced

See also: DBGOTO()

See Also: DBGOTO()

AT()
Return the position of a substring within a character string

Syntax

AT(<cSearch>, <cTarget>) --> nPosition

Arguments

<cSearch> is the character substring for which to search.

<cTarget> is the character string to search.

Returns

AT() returns the position of the first instance of <cSearch> within <cTarget> as an integer numeric value. If <cSearch> is not found, AT() returns zero.

Description

AT() is a character function used to determine the position of the first occurrence of a character substring within another string. If you only need to know whether a substring exists within another string, use the $ operator. To find the last instance of a substring within a string, use RAT().

Examples

. These examples show typical use of AT():

```
? AT("a", "abcde")                  // Result: 1
? AT("bcd", "abcde")                // Result: 2
? AT("a", "bcde")                   // Result: 0
```

. This example splits a character string based on the position of a comma within the target string:

```
cTarget := "Langtree, Lilly"
? SUBSTR(cTarget, 1, AT(",", cTarget) - 1)
// Result: Langtree

? SUBSTR(cTarget, AT(",", cTarget) + 2)
// Result: Lilly
```

Files: Library is CLIPPER.LIB.

See Also: RAT() STRTRAN() SUBSTR()

SEEK command
Search an Order for a specified key value

Syntax

SEEK <expSearch> **[SOFTSEEK]**

Arguments

<expSearch> is an expression to match with an Order key value.

SOFTSEEK causes the record pointer to be moved to the next record with a higher key value after a failed Order search. Default behavior moves the record pointer to EOF() after a failed Order search.

Description

SEEK is a database command that searches the controlling Order from the first key value and proceeding until a match is found or there is a key value greater than <expSearch>. If there is a match, the record pointer is positioned to the identity found in the Order. If SOFTSEEK is OFF (the default) and SEEK does not find a match, the record pointer is positioned to LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.).

SOFTSEEK enables a method of searching an Order and returning a record even if there is no match for a specified key.

When SOFTSEEK is ON and a match for a SEEK is not found, the record pointer is set to the next record in the Order with a higher key value than the SEEK argument. Records not visible because SET FILTER and/or SET DELETED are skipped when searching for the next higher key value. If there is no record with a higher key value, the record pointer is positioned at LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.). FOUND() returns true (.T.) only if the record is actually found. FOUND() never returns true (.T.) for a relative find.

When SOFTSEEK is OFF and a SEEK is unsuccessful, the record pointer is positioned at LASTREC() + 1, EOF() returns true (.T.), and FOUND() returns false (.F.).

SEEK with the SOFTSEEK clause is, effectively, the same as performing SET SOFTSEEK then SEEK in earlier versions of CA-Clipper except that is does not change the global setting of SOFTSEEK.

Examples

. The following example searches for "Doe" using the SEEK command:

```
USE Customer NEW
SET ORDER TO Customer

? SET( _SET_SOFTSEEK )   // (.F.)
SEEK "Doe"
? SET( _SET_SOFTSEEK )   // Still (.F.)

IF FOUND()
   .
   . < statements >
   .
ENDIF
```

. The following example performs a soft seek for "Doe" using SOFTSEEK clause of the SEEK command:

```
USE Customer NEW
```

```
SET ORDER TO Customer

SEEK "Doe" SOFTSEEK

IF !FOUND()
   ? Customer->Name      // Returns next logical name after "Doe"
ENDIF
```

Status: Enhanced

See also: DBSEEK()

See Also: DBSEEK()

SET INDEX command
Open one or more Order Bags in the current work area

Syntax

SET INDEX TO **[<xcOrderBagName list>] [ADDITIVE]**

Arguments

<cOrderBagName list> specifies Order Bags to be emptied into the Order List of the current work area.

ADDITIVE adds Order Bags to an existing Order List.

Description

By default, SET INDEX, without the ADDITIVE clause, clears the currently active Order List, then constructs a new Order List from the Orders in the specified Order Bags in the current work area. When several Order Bags are opened, the first Order in the first Order Bag becomes the controlling Order (has focus). The record pointer is initially positioned at the first logical identity in this Order.

If an Order List exists when you SET INDEX ... ADDITIVE, the Orders in the new Order Bag are added to the end of the Order List. The previous controlling Order continues to be the controlling Order.

If no Order List exists when you SET INDEX ... ADDITIVE, the first Order in the first Order Bag in <cOrderBagName list> becomes the controlling Order.

During database processing, all open Orders are updated whenever a key value is appended or changed, unless the Order was created using a scoping condition and the key value does not match. To change the controlling Order without issuing another SET INDEX command, use SET ORDER or ORDSETFOCUS(). To add Orders without closing the currently open Orders, use the ADDITIVE clause.

Examples

. This example opens a database and its associated indexes:

```
USE Sales NEW
SET INDEX TO Sales, Sales1, Sales2
```

. This example opens an index without closing any indexes that are already open:

```
SET INDEX TO Sales3 ADDITIVE
```

Status: Enhanced

See also: CLOSE, DBCLEARINDEX(), DBSETINDEX(), INDEX, REINDEX, SET ORDER, USE

See Also: CLOSE DBCLEARIND() DBSETINDEX()* REINDEX USE

SET ORDER command
Select the controlling Order

Syntax

SET ORDER TO **[<nOrder> | [TAG <cOrderName>]**
[IN <xcOrderBagName>]]

Arguments

TAG is an optional clause that provides compatibility with RDDs that access multiple-Order Order Bags. You must use this keyword anytime you specify <cOrderName>.

<cOrderName> is the name of an Order, a logical arrangement of a database according to a keyed pair. This Order will become the controlling Order in the Order List. If you specify <cOrderName>, you must use the keyword TAG.

Note: This differs from dBASE and FoxPro where TAG is totally optional.

<nOrder> is the number of the target Order in the Order List.

You may represent the Order as an integer or as a character string enclosed in quotes.

IN <xcOrderBagName> is the name of a disk file containing one or more Orders. You may specify <xcOrderBagName> as the filename with or without the pathname or appropriate extension. If you do not include the extension as part of <xcOrderBagName> CA-Clipper uses the default extension of the current RDD.

Description

When you SET ORDER TO a new controlling Order (index), all Orders are properly updated when you either append or edit records. This is true even if you SET ORDER TO 0. After a change of controlling Order, the record pointer still points to the same record.

SET ORDER TO 0 restores the database access to natural order, but leaves all Orders open. SET ORDER TO with no arguments closes all Orders and empties the Order List

Though you may use <cOrderName> or <nOrder> to specify the target Order, <nOrder> is only provided for compatibility with earlier versions of CA-Clipper. Using <cOrderName> is a surer way of accessing the correct Order in the Order List.

If you supply <xcOrderBagName> only the Orders belonging to <xcOrderBagName> in the Order List are searched. Usually, you need not specify <xcOrderBagName> if you use unique Order Names throughout an application.

To determine which Order is the controlling Order use the ORDSETFOCUS() function.

In RDDs that support production or structural indexes (e.g., DBFCDX, DBPX), if you specify a Tag but do not specify an Order Bag, the Tag is created and added to the index. If no production or structural index exists, it will be created and the Tag will be added to it. When using RDDs that support multiple Order Bags, you must explicitly SET ORDER (or ORDSETFOCUS()) to the desired controlling Order. If you do not specify a controlling Order, the data file will be viewed in natural Order.

SET ORDER can open Orders in a network environment instead of the INDEX clause of the USE command. Generally, specify USE, then test to

```
determine whether the USE succeeded.  If it did succeed, open the
associated Orders with SET ORDER.  See the example below.
```

Examples

```
USE Customer NEW
IF (! NETERR())
   SET ORDER TO Customer
ENDIF

SET ORDER TO "CuAcct"         // CuAcct is an Order in Customer
```

Status: Enhanced

See also: INDEX, INDEXORD(), SEEK, SET INDEX, USE

See Also: INDEX INDEXORD() SEEK SET INDEX USE

Inkey Codes

Cursor Movement Keys

Uparrow, Ctrl-E	5	K_UP
Dnarrow, Ctrl-X	24	K_DOWN
Leftarrow, Ctrl-S	19	K_LEFT
Rightarrow, Ctrl-D	4	K_RIGHT
Home, Ctrl-A	1	K_HOME
End, Ctrl-F	6	K_END
PgUp, Ctrl-R	18	K_PGUP
PgDn, Ctrl-C	3	K_PGDN
Ctrl-Leftarrow, Ctrl-Z	26	K_CTRL_LEFT
Ctrl-Rightarrow, Ctrl-B	2	K_CTRL_RIGHT
Ctrl-Home, Ctrl-]	29	K_CTRL_HOME
Ctrl-End, Ctrl-W	23	K_CTRL_END
Ctrl-PgUp, Ctrl-Hyphen	31	K_CTRL_PGUP
Ctrl-PgDn, Ctrl-^	30	K_CTRL_PGDN
Ctrl-Return	10	K_CTRL_RET
Ctrl-ScrollLock	3	K_CTRL_C
Esc, Ctrl-[	27	K_ESC
Return, Ctrl-M	13	K_RETURN
Enter, Ctrl-M	13	K_ENTER

Editing Keys

Del, Ctrl-G	7	K_DEL
Tab, Ctrl-I	9	K_TAB
Shift-Tab	271	K_SH_TAB
Ins, Ctrl-V	22	K_INS
Backspace, Ctrl-H	8	K_BS
Ctrl-Backspace	127	K_CTRL_BS

Function Keys

F1, Ctrl-\	28	K_F1
F2	-1	K_F2
F3	-2	K_F3
F4	-3	K_F4
F5	-4	K_F5
F6	-5	K_F6
F7	-6	K_F7
F8	-7	K_F8
F9	-8	K_F9
F10	-9	K_F10

Alt and Ctrl Keys

Alt-A	286	K_ALT_A	Alt-N	305	K_ALT_N	Ctrl-A	1	K_CTRL_A	Ctrl-N	14	K_CTRL_N
Alt-B	304	K_ALT_B	Alt-O	280	K_ALT_O	Ctrl-B	2	K_CTRL_B	Ctrl-O	15	K_CTRL_O
Alt-C	302	K_ALT_C	Alt-P	281	K_ALT_P	Ctrl-C	3	K_CTRL_C	Ctrl-P	16	K_CTRL_P
Alt-D	288	K_ALT_D	Alt-Q	272	K_ALT_Q	Ctrl-D	4	K_CTRL_D	Ctrl-Q	17	K_CTRL_Q
Alt-E	274	K_ALT_E	Alt-R	275	K_ALT_R	Ctrl-E	5	K_CTRL_E	Ctrl-R	18	K_CTRL_R
Alt-F	289	K_ALT_F	Alt-S	287	K_ALT_S	Ctrl-F	6	K_CTRL_F	Ctrl-S	19	K_CTRL_S
Alt-G	290	K_ALT_G	Alt-T	276	K_ALT_T	Ctrl-G	7	K_CTRL_G	Ctrl-T	20	K_CTRL_T
Alt-H	291	K_ALT_H	Alt-U	278	K_ALT_U	Ctrl-H	8	K_CTRL_H	Ctrl-U	21	K_CTRL_U
Alt-I	279	K_ALT_I	Alt-V	303	K_ALT_V	Ctrl-I	9	K_CTRL_I	Ctrl-V	22	K_CTRL_V
Alt-J	292	K_ALT_J	Alt-W	273	K_ALT_W	Ctrl-J	10	K_CTRL_J	Ctrl-W	23	K_CTRL_W
Alt-K	293	K_ALT_K	Alt-X	301	K_ALT_X	Ctrl-K	11	K_CTRL_K	Ctrl-X	24	K_Ctrl_X
Alt-L	294	K_ALT_L	Alt-Y	277	K_ALT_Y	Ctrl-L	12	K_CTRL_L	Ctrl-Y	25	K_Ctrl_Y
Alt-M	306	K_ALT_M	Alt-Z	300	K_ALT_Z	Ctrl-M	13	K_CTRL_M	Ctrl-Z	26	K_CTRL_Z

Shift, Alt, and Ctrl-Function Keys

Shift-F1	-10	K_SH_F1	Ctrl-F1	-20	K_CTRL_F1	Alt-F1	-30	K_ALT_F1
Shift-F2	-11	K_SH_F2	Ctrl-F2	-21	K_CTRL_F2	Alt-F2	-31	K_ALT_F2
Shift-F3	-12	K_SH_F3	Ctrl-F3	-22	K_CTRL_F3	Alt-F3	-32	K_ALT_F3
Shift-F4	-13	K_SH_F4	Ctrl-F4	-23	K_CTRL_F4	Alt-F4	-33	K_ALT_F4
Shift-F5	-14	K_SH_F5	Ctrl-F5	-24	K_CTRL_F5	Alt-F5	-34	K_ALT_F5
Shift-F6	-15	K_SH_F6	Ctrl-F6	-25	K_CTRL_F6	Alt-F6	-35	K_ALT_F6
Shift-F7	-16	K_SH_F7	Ctrl-F7	-26	K_CTRL_F7	Alt-F7	-36	K_ALT_F7
Shift-F8	-17	K_SH_F8	Ctrl-F8	-27	K_CTRL_F8	Alt-F8	-37	K_ALT_F8
Shift-F9	-18	K_SH_F9	Ctrl-F9	-28	K_CTRL_F9	Alt-F9	-38	K_ALT_F9
Shift-F10	-19	K_SH_F10	Ctrl-F10	-29	K_CTRL_F10	Alt-F10	-39	K_ALT_F10

Colors

Color			Monochrome	Color			Monochrome
Black, Space	N	0	Black	Gray	N+	8	Black
Blue	B	1	Underline	Bright Blue	B+	9	Bright Underline
Green	G	2	White	Bright Green	G+	10	Bright White
Cyan	BG	3	White	Bright Cyan	BG+	11	Bright White
Red	R	4	White	Bright Red	R+	12	Bright White
Magenta	RB	5	White	Bright Magenta	RB+	13	Bright White
Brown	GR	6	White	Yellow	GR+	14	Bright White
White	W	7	White	Bright White	W+	15	Bright White

Black	U	Underline
Inverse	I	Inverse video
Black	X	Blank

. Note

If numbers are used and the background color is greater than 7, the foreground color is displayed blinking. If letters are used and the high intensity attribute is applied to the background color, the foreground color is displayed in high intensity.

See Also: SET COLOR* SETCOLOR()

Picture Codes

Say Picture Functions

B	Display numbers left-justified	X	Display DB after negative numbers
C	Display CR after positive numbers	Z	Display zeros as blanks
D	Display date in SET DATE format	(	Enclose negative numbers in parens
E	Display date in British format	!	Convert alpha characters to upper
R	Insert non-template characters		

Say Picture Template Symbols

A	Display digits for any data type	!	Convert alpha character to upper
N	"	$	Display dollar sign instead of
X	"		leading space in a numeric
9	"	*	Display asterisk instead of
#	"		leading space in a numeric
L	Display logicals as "T" or "F"	.	Specify a decimal point position
Y	Display logicals as "Y" or "N"	,	Specify a comma position

Get Picture Functions

A	C	Allow only alpha characters
B	N	Display numbers left-justified
C	N	Display CR after positive numbers
D	D,N	Display dates in SET DATE format
E	D,N	Display dates with day and month inverted independent of the current DATE SETting, numerics with comma and period reverse
K	All	Delete default text if first key is not a cursor key
R	C	Insert non-template characters in the display but do not save in the Get variable
S<n>	C	Allows horizontal scrolling within a Get. <n> is an integer that specifies the width of the region
X	N	Display DB after negative numbers
Z	N	Display zero as blanks
(	N	Display negative numbers in parentheses with leading spaces
)	N	Display negative numbers in parentheses without leading spaces
!	C	Convert alphabetic character to upper case

Get Picture Template Symbols

A	Allow only alphabetic characters
N	Allow only alphabetic and numeric characters
X	Allow any character
9	Allow digits for any data type including sign for numerics
#	Allow digits, signs and spaces for any data type
L	Allow only T, F, Y or N
Y	Allow only Y or N
!	Convert alphabetic character to upper case
$	Display a dollar sign in place of a leading space in a numeric
*	Display an asterisk in place of a leading space in a numeric
.	Display a decimal point
,	Display a comma

ASCII Chart

Dec	Hex	Char		Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00h		NUL	64	40h	@	128	80h	.	192	C0h	+
1	01h	.	SOH	65	41h	A	129	81h	.	193	C1h	-
2	02h	.	STX	66	42h	B	130	82h	.	194	C2h	-
3	03h	.	ETX	67	43h	C	131	83h	.	195	C3h	|
4	04h	.	EOT	68	44h	D	132	84h	.	196	C4h	-
5	05h	.	ENQ	69	45h	E	133	85h	.	197	C5h	+
6	06h	.	ACK	70	46h	F	134	86h	.	198	C6h	-
7	07h	.	BEL	71	47h	G	135	87h	.	199	C7h	-
8	08h	.	BS	72	48h	H	136	88h	.	200	C8h	+
9	09h	.	HT	73	49h	I	137	89h	.	201	C9h	+
10	0Ah	.	LF	74	4Ah	J	138	8Ah	.	202	CAh	-
11	0Bh	.	VT	75	4Bh	K	139	8Bh	.	203	CBh	-
12	0Ch	.	FF	76	4Ch	L	140	8Ch	.	204	CCh	|
13	0Dh	.	CR	77	4Dh	M	141	8Dh	.	205	CDh	-
14	0Eh	.	SO	78	4Eh	N	142	8Eh	.	206	CEh	+
15	0Fh	.	SI	79	4Fh	O	143	8Fh	.	207	CFh	-
16	10h	.	DLE	80	50h	P	144	90h	.	208	D0h	-
17	11h	.	DC1	81	51h	Q	145	91h	.	209	D1h	-
18	12h	.	DC2	82	52h	R	146	92h	.	210	D2h	-
19	13h	.	DC3	83	53h	S	147	93h	.	211	D3h	+
20	14h	.	DC4	84	54h	T	148	94h	.	212	D4h	+
21	15h	.	NAK	85	55h	U	149	95h	.	213	D5h	+
22	16h	.	SYN	86	56h	V	150	96h	.	214	D6h	+
23	17h	.	ETB	87	57h	W	151	97h	.	215	D7h	+
24	18h	.	CAN	88	58h	X	152	98h	.	216	D8h	+
25	19h	.	EM	89	59h	Y	153	99h	.	217	D9h	+
26	1Ah	.	SUB	90	5Ah	Z	154	9Ah	.	218	DAh	+
27	1Bh	.	ESC	91	5Bh	[	155	9Bh	.	219	DBh	#
28	1Ch	.	FS	92	5Ch	\	156	9Ch	.	220	DCh	#
29	1Dh	.	GS	93	5Dh	]	157	9Dh	.	221	DDh	#
30	1Eh	.	RS	94	5Eh	^	158	9Eh	.	222	DEh	#
31	1Fh	.	US	95	5Fh	_	159	9Fh	.	223	DFh	#
32	20h			96	60h	`	160	A0h	.	224	E0h	.
33	21h	!		97	61h	a	161	A1h	.	225	E1h	.
34	22h	"		98	62h	b	162	A2h	.	226	E2h	.
35	23h	#		99	63h	c	163	A3h	.	227	E3h	.
36	24h	$		100	64h	d	164	A4h	.	228	E4h	.
37	25h	%		101	65h	e	165	A5h	.	229	E5h	.
38	26h	&		102	66h	f	166	A6h	.	230	E6h	.
39	27h	'		103	67h	g	167	A7h	.	231	E7h	.
40	28h	(		104	68h	h	168	A8h	.	232	E8h	.
41	29h	)		105	69h	i	169	A9h	.	233	E9h	.
42	2Ah	*		106	6Ah	j	170	AAh	.	234	EAh	.
43	2Bh	+		107	6Bh	k	171	ABh	.	235	EBh	.
44	2Ch	,		108	6Ch	l	172	ACh	.	236	ECh	.
45	2Dh	-		109	6Dh	m	173	ADh	.	237	EDh	.
46	2Eh	.		110	6Eh	n	174	AEh	.	238	EEh	.
47	2Fh	/		111	6Fh	o	175	AFh	.	239	EFh	.
48	30h	0		112	70h	p	176	B0h	#	240	F0h	.
49	31h	1		113	71h	q	177	B1h	#	241	F1h	.
50	32h	2		114	72h	r	178	B2h	#	242	F2h	.
51	33h	3		115	73h	s	179	B3h	|	243	F3h	.
52	34h	4		116	74h	t	180	B4h	|	244	F4h	.
53	35h	5		117	75h	u	181	B5h	|	245	F5h	.
54	36h	6		118	76h	v	182	B6h	|	246	F6h	.
55	37h	7		119	77h	w	183	B7h	+	247	F7h	.
56	38h	8		120	78h	x	184	B8h	+	248	F8h	.
57	39h	9		121	79h	y	185	B9h	|	249	F9h	.
58	3Ah	:		122	7Ah	z	186	BAh	|	250	FAh	.
59	3Bh	;		123	7Bh	{	187	BBh	+	251	FBh	.
60	3Ch	<		124	7Ch	|	188	BCh	+	252	FCh	.
61	3Dh	=		125	7Dh	}	189	BDh	+	253	FDh	.

62	3Eh	>	|	126	7Eh	~	|	190	BEh	+	|	254	FEh	.
63	3Fh	?	|	127	7Fh	.	|	191	BFh	+	|	255	FFh	.

ATAIL()
Return the highest numbered element of an array

Syntax

ATAIL(<aArray>) --> Element

Arguments

<aArray> is the array.

Returns

ATAIL() returns either a value or a reference to an array or object. The array is not changed.

Description

ATAIL() is an array function that returns the highest numbered element of an array. It can be used in applications as shorthand for <aArray>[LEN(<aArray>)] when you need to obtain the last element of an array.

Examples

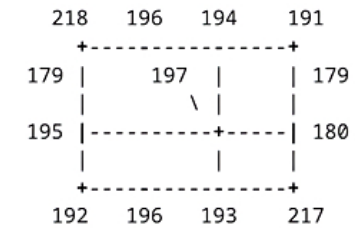
. The following example creates a literal array and returns that last element of the array:

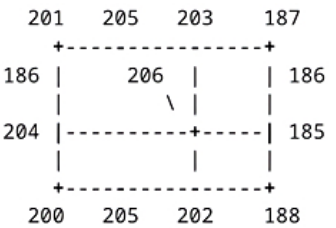
```
aArray := {"a", "b", "c", "d"}
? ATAIL(aArray)                    // Result: d
```

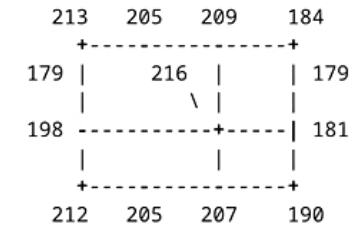
Files: Library is CLIPPER.LIB.

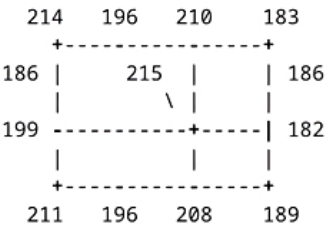
See Also: LEN()

Box Characters









Special Characters

Arrows

. 30	. 24	. 18
. 31	. 25	. 23
. 16	. 26	
. 17	. 27	. 29

Blocks

# 176	# 219	
	# 221	
# 177	# 220	
	. 222	
# 178	# 223	

Currency

. 155	Cents	
. 156	Pounds	
. 157	Yen	
. 159	Francs	
$ 36	Dollars	

Foreign

131 .	130 .	. 144
132 .	. 142	140 .
133 .	139 .	
160 .	147 .	
134 .	. 143	148 .
145 .	. 146	149 .
135 .	. 128	162 .
136 .	164 .	. 165
137 .	141 .	
138 .	161 .	

150 .		
129 .	. 154	
151 .		
163 .		
152 .		
225 .		

Greek

. 224		Alpha	a
. 225		Beta	b
. 226		Gamma	g
. 235		Delta	d
. 233		Theta	th
. 230		Mu	m
. 227		Pi	p
. 229	. 228	Sigma	s
. 231		Tau	t
. 237	. 232	Phi	ph
. 234		Omega	

Mathematical

. 143		. 246	Division
. 238	Is an element of	. 247	Nearly equal
. 239	Intersection	. 248	Degree
. 240	Identical to, congruent	. 249	Multiplied by
. 241	Plus or minus	. 250	
. 242	Greater than or equal to	. 251	Square root
. 243	Less than or equal to	. 252	Nth power
		. 253	Squared (second power)
		. 236	Infinity
. 244	Integral	. 244	Integral
. 245		| 179	
		. 245	

Bullets

. 7	. 5	. 166
. 4	. 6	. 167
. 249	. 11	. 168
. 250	. 12	. 173
. 3	* 42	. 174

Other

. 175	
. 19	
. 20	Paragraph or Return character
. 21	Section number
. 158	

Reserved Words, Identifiers and Functions

AADD	ABS	ACHOICE	ACLONE	ACOPY	ADEL
ADIR	AEVAL	AFIELDS	AFILL	AINS	ALERT
ALIAS	ALLTRIM	ALTD	ARRAY	ASC	ASCAN
ASIZE	ASORT	AT	ATAIL		

BIN2I	BIN2L	BIN2W	BREAK	BOF

CDOW	CHR	CMONTH	COL
COLORSELECT	CTOD	CURDIR	

DATE	DAY	DBAPPEND
DBCLEARFILTER	DBCLEARINDEX	DBCLEARRELATION
DBCLOSEALL	DBCLOSEAREA	DBCOMMIT
DBCOMMITALL	DBCREATE	DBCREATEINDEX
DBDELETE	DBEDIT	DBEVAL
DBFILTER	DBFONLY	DBGOBOTTOM
DBGOTO	DBGOTOP	DBGSHADOW
DBRECALL	DBREINDEX	DBRELATION
DBRSELECT	DBSEEK	DBSELECTAREA
DBSETDRIVER	DBSETFILTER	DBSETINDEX
DBSETORDER	DBSETRELATION	DBSKIP
DBSTRUCT	DBUNLOCK	DBUNLOCKALL
DBUSEAREA	DEFPATH	DELETED
DELIM	DESCEND	DEVOUT
DEVOUTPICT	DEVPPOS	DIRECTORY
DISKSPACE	DISPBEGIN	DISPBOX
DISPCOUNT	DISPEND	DISPOUT
DISPOUTAT	DO	DOSERROR
DOW	DTOC	DTOS

ELSE	ELSEIF	END
ENDCASE	ENDDO	ENDIF
EMPTY	EOF	ERRORBLOCK
ERRORINHANDLER	ERRORLEVEL	ERRORNEW
ERRORSYS	EXP	

FCLOSE	FCOUNT	FCREATE
FERASE	FERROR	FIELDBLOCK
FIELDNAME	FIELDGET	FIELDPOS
FIELDPUT	FIELDWBLOCK	FILE
FLOCK	FOPEN	FOUND
FREAD	FREADSTR	FRENAME
FSEEK	FWRITE	FUNCTION

GETACTIVE	GETAPPLYKEY	GETDOSETKEY
GETE	GETENV	GETNEW
GETPOSTVALIDATE	GETPREVALIDATE	GETREADER

HARDCR	HEADER		
I2BIN	IF	IIF	INDEXEXT
INDEXKEY	INDEXORD	INKEY	INT
ISALPHA	ISCOLOR	ISDIGIT	ISLOWER
ISPRINTER	ISUPPER		
L2BIN	LASTKEY	LASTREC	LEFT
LEN	LOCK	LOG	LOWER
LTRIM	LUPDATE		
MAX	MAXCOL	MAXROW	MEMOEDIT
MEMOLINE	MEMOREAD	MEMORY	MEMOTRAN
MEMOWRIT	MEMVARBLOCK	MIN	MLCOUNT
MLCTOPOS	MLPOS	MONTH	MPOSTOLC
NETERR	NETNAME	NEXTKEY	NOSNOW
ORDBAGEXT	ORDBAGNAME	ORDCONDSET	
ORDCREATE	ORDDESTROY	ORDFOR	
ORDKEY	ORDLISTADD	ORDLISTCLEAR	
ORDLISTREBUILD	ORDNAME	ORDNUMBER	
ORDSETFOCUS	OS	OUTERR	
OUTSTD			
PARSEHEADER	PAD	PADC	PADL
PADR	PCOL	PCOUNT	PROCLINE
PROCEDURE	PROCNAME	PROW	
QOUT	QQOUT		
RANGECHECK	RAT	RDDLIST	
RDDNAME	RDDSYS	RDDREGISTER	
RDDSETDEFAULT	READEXIT	READFORMAT	
READINSERT	READKEY	READKILL	
READMODAL	READUPDATED	READVAR	
RECCOUNT	RECNO	RECSIZE	
REPLICATE	RESTSCREEN	RIGHT	
RLOCK	ROUND	ROW	
RTRIM			
SAVESCREEN	SCROLL	SDF	SECONDS
SELECT	SET	SETBLINK	SETCANCEL
SETCOLOR	SETCURSOR	SETKEY	SETMODE
SETPOS	SETPRC	SETTYPEAHEAD	SOUNDEX
SPACE	SQRT	STR	STRTRAN
STUFF	SUBSTR	SYSINIT	

```
    |                                                    |
    |  TBCOLUMNNEW  TBROWSEDB   TBROWSENEW  TIME        |
    |  TONE         TRANSFORM   TRIM        TYPE        |
    |                                                    |
    |----------------------------------------------------|
    |                                                    |
    |  UPDATED  UPPER  USED                              |
    |                                                    |
    |----------------------------------------------------|
    |                                                    |
    |  VAL  VALTYPE   VERSION                            |
    |                                                    |
    |----------------------------------------------------|
    |                                                    |
    |  WHILE  WORD                                       |
    |                                                    |
    |----------------------------------------------------|
    |                                                    |
    |  YEAR                                              |
    |                                                    |
    +----------------------------------------------------+
```

. **Notes**

. **Reserved words** cannot be used for variable, procedure, or user-defined function names.

. **Reserved functions** are built into the compiler and the efore cannot be redefined by an application.

. **Abbreviations** of reserved words and functions of four or more characters are also reserved.

. All identifiers that begin with one or more underscore characters (_) are designated as system identifiers and are therefore reserved.

. All reserved words, identifiers and functions are case in-sensitive (not case sensitive) and are significant for the first ten (10) characters.

---

dBASE Commands and Functions Not Supported by CA-Clipper

Some dBASE commands and functions are not supported by CA-Clipper. CA-Clipper does not support any of the commands that are used primarily in the interactive or "dot prompt" mode. In the interactive mode, you may instantly query the various databases without writing a program, however, CA-Clipper has been designed to compile and execute programs significantly faster than can be accomplished in the interactive mode. The dBASE commands and functions that are not supported by CA-Clipper are listed in the table below.

CA-Clipper Equivalents of dBASE Commands & Functions:

dBASE Command/Function	CA-Clipper Equivalent
APPEND	DBU.EXE
ASSIST	DBU.EXE
BROWSE	BROWSE(), DBEDIT(), TBrowse Class
CHANGE	DBU.EXE
CLEAR FIELDS	n/a
CREATE/MODIFY LABEL	RL.EXE
CREATE/MODIFY QUERY	n/a
CREATE/MODIFY REPORT	RL.EXE
CREATE/MODIFY SCREEN	n/a
CREATE/MODIFY STRUCTURE	DBU.EXE
CREATE/MODIFY VIEW	DBU.EXE
EDIT	DBU.EXE
ERROR()	Error:genCode, Error:osCode, Error:SubCode messages
EXPORT TO	n/a
HELP	The Guide To CA-Clipper
IMPORT FROM	n/a
INSERT	n/a
LIST/DISPLAY FILES	DBU.EXE
LIST/DISPLAY HISTORY	The CA-Clipper Debugger
LIST/DISPLAY MEMORY	The CA-Clipper Debugger
LIST/DISPLAY STATUS	The CA-Clipper Debugger
LIST/DISPLAY STRUCTURE	The CA-Clipper Debugger
LOAD	RTLINK.EXE
LOGOUT	n/a
MESSAGE()	Error:description message
MODIFY COMMAND	PE.EXE
ON ERROR	ERRORBLOCK()
ON ESCAPE	SET KEY User function
ON KEY	SET KEY User function
RESUME	RETURN false (.F.) from an error handling block if Error:canDefault contains true (.T.)
RETRY	RETURN true (.T.) from an error handling block if Error:canRetry contains true (.T.)
RETURN TO MASTER	BEGIN SEQUENCE...BREAK...END
SET	The CA-Clipper Debugger
SET CARRY	n/a
SET CATALOG	n/a
SET COLOR ON | OFF	n/a
SET DEBUG	ALTD()
SET DOHISTORY	The CA-Clipper Debugger
SET ECHO	The CA-Clipper Debugger
SET ENCRYPTION	n/a
SET FIELDS	DBU.EXE
SET HEADING	n/a
SET HELP	n/a
SET HISTORY	n/a
SET MEMOWIDTH	MEMOLINE(), MEMOEDIT(), MLCOUNT()
SET MENUS	n/a
SET MESSAGE	n/a
SET SAFETY	n/a

```
 SET STATUS                | n/a
 SET TALK                  | The CA-Clipper Debugger
 SET TITLE                 | n/a
 SET VIEW                  | DBU.EXE
-------------------------------------------------------------------------------
```

BIN2I()
Convert a 16-bit signed integer to a numeric value

Syntax

BIN2I(<cSignedInt>) --> nNumber

Arguments

<cSignedInt> is a character string in the form of a 16-bit signed integer number--least significant byte first. Only the first two characters are used by the function; all others are ignored.

Returns

BIN2I() returns an integer numeric value.

Description

BIN2I() is a low-level file function that is used with FREAD() to convert a two-byte character string formatted as a signed integer to a CA-Clipper numeric data type. This is most useful when you are reading foreign file types and want to read numeric data in its native format.

Examples

. This example opens a database file using low-level file functions and reads the date of last update (bytes 1-3). The result is the same as with LUPDATE():

```
#include "Fileio.ch"
//
nHandle := FOPEN("Sales.dbf", FO_READ)
//
// Point to byte 1 in the file
FSEEK(nHandle, 1, FS_SET)
//
// Read date of last update
nYear := BIN2I(FREADSTR(nHandle, 1) + CHR(0))
nMonth := BIN2I(FREADSTR(nHandle, 1) + CHR(0))
nDay := BIN2I(FREADSTR(nHandle, 1) + CHR(0))
//
? LTRIM(STR(nMonth)), LTRIM(STR(nDay)), LTRIM(STR(nYear))
FCLOSE(nHandle)
```

Files: Library is EXTEND.LIB, source file is SOURCE\SAMPLE\EXAMPLEA.ASM.

See Also: BIN2L() BIN2W() FREAD() FREADSTR() I2BIN() L2BIN()